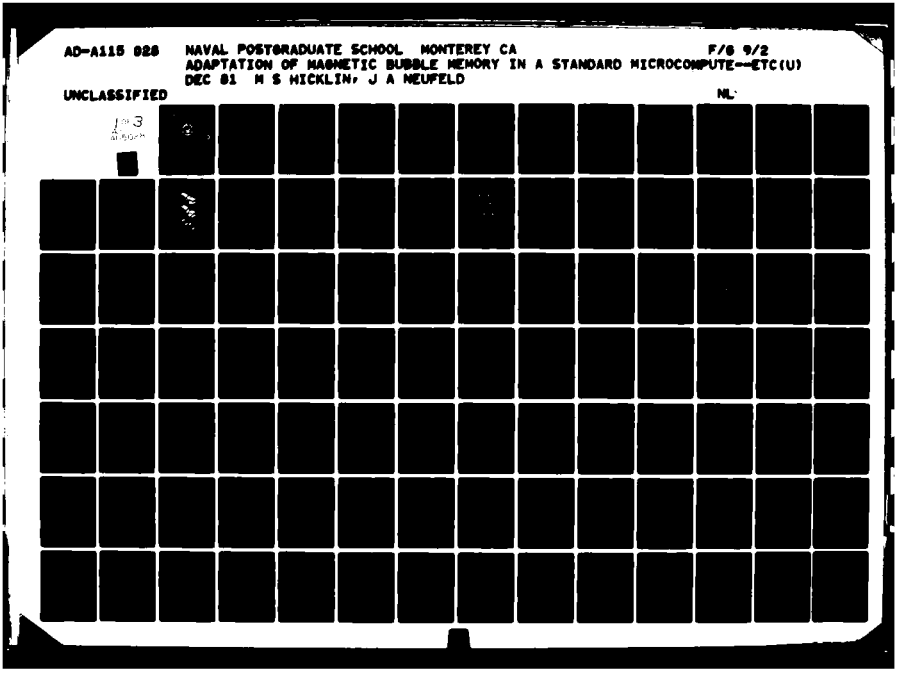
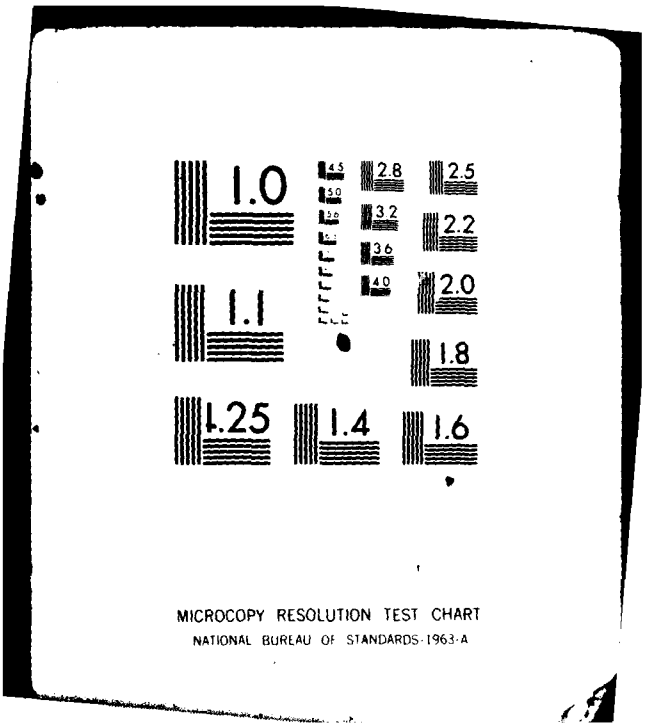
AD-A115 928 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
ADAPTATION OF MAGNETIC BUBBLE MEMORY IN A STANDARD MICROCOMPUTE--ETC(U)
DEC 81 M S HICKLIN, J A NEUFELD

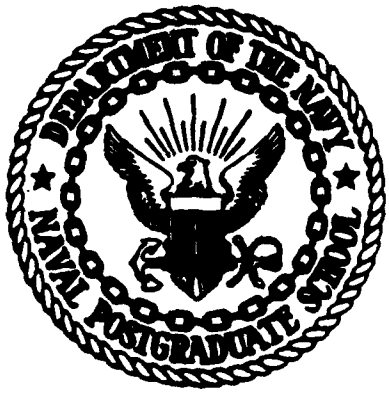
UNCLASSIFIED NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0
11.2
12.5
14.0
16.0
18.0
20.0
22.4
25.0
28.0
32.0
36.0
40.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

ADAPTATION OF MAGNETIC BUBBLE MEMORY
IN A STANDARD MICROCOMPUTER ENVIRONMENT

by

Michael S. Hicklin

and

Jeffrey A. Neufeld

December 1981

Thesis Advisor: R. R. Stilwell

Approved for public release; distribution unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A115028	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Adaptation of Magnetic Bubble Memory in a Standard Microcomputer Environment		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; December 1981
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael S. Hicklin Jeffrey A. Neufeld		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December 1981
		13. NUMBER OF PAGES 198
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Magnetic bubble memory, microcomputer operating system, CP/M-86, secondary storage media

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Magnetic bubble memory is a new digital storage technology that offers many significant advantages over currently existing secondary storage media. Bubble memories, with high densities and relatively fast access times, are non-volatile semiconductor devices that provide a high degree of reliability in harsh environments. This technology has the potential for a vital and unique role in

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(continuation of abstract)

both the civilian and military computing environments due to the combination of characteristics exhibited by magnetic domain devices.

This thesis presents an implementation of a magnetic bubble device utilizing a conventional operating system, Digital Research's CP/M-86, and a standard commercial 16-bit microcomputer, the Intel iSBC 86/12A. A fully operational system capable of testing, evaluating and utilizing a magnetic bubble device in a standard user environment is presented.

Approved for public release; distribution unlimited.

Adaptation of Magnetic Bubble Memory in a Standard Microcomputer Environment

by

Michael S. Hicklin
Captain, United States Marine Corps
B.S.M.E., University of Utah

Jeffrey A. Neufeld
Captain, United States Marine Corps
B.S.S.E., United States Naval Academy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1981

Authors: Michael S. Hicklin

Jeffrey A. Neufeld

Approved by:

Thesis Advisor

Second Reader

Chairman, Department of Computer Sciences

Dean of Information and Policy Sciences

ABSTRACT

Magnetic bubble memory is a new digital storage technology that offers many significant advantages over currently existing secondary storage media. Bubble memories, with high densities and relatively fast access times, are non-volatile semiconductor devices that provide a high degree of reliability in harsh environments. This technology has the potential for a vital and unique role in both the civilian and military computing environments due to the combination of characteristics exhibited by magnetic domain devices.

This thesis presents an implementation of a magnetic bubble device utilizing a conventional operating system, Digital Research's CP/M-86, and a standard commercial 16-bit microcomputer, the Intel iSBC 86/12A. A fully operational system capable of testing, evaluating and utilizing a magnetic bubble device in a standard user environment is presented.

TABLE OF CONTENTS

DISCLAIMER

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis will be listed below, following the firm holding the trademark.

Intel Corporation, Santa Clara, California:

Intel	MULTIBUS	INTELLEC MDS
Intel 8080	Intel 8086	iSBC 86/12A
iSBC 202	i8259	

Pacific Cyber/Metrixs Incorporated, Dublin, California:

Bubbl-Tec	Bubbl-Machine	MBB-80 Bubbl-Board

Digital Research, Pacific Grove, California:

CP/M-80	CP/M-86	CP/M

I. INTRODUCTION

Magnetic bubble memory is a new digital storage technology that offers many significant advantages over currently existing secondary storage mediums. Bubble memories, with high densities and relatively fast access times, are non-volatile semiconductor devices that provide a high degree of reliability in harsh environments. This technology has the potential for a vital and unique role in both the civilian and military computing environments due to the combination of characteristics exhibited by magnetic domain devices.

This thesis presents an implementation of a magnetic bubble device (MBB-80) utilizing a conventional operating system (CP/M-86) and a commercial 16-bit microprocessor (Intel 8086). A fully operational system capable of testing, evaluating, and utilizing a magnetic bubble device in a standard user environment is presented.

There are four major phases into which this thesis is organized. The first phase will present an overview of bubble domain devices to provide an understanding and evaluation of their potential applications as mass storage mediums. Chapter II will describe the theory of magnetic

bubble devices and the current state of magnetic domain technology. Chapter III will present an evaluation of bubble memory technology and utilization along with a justification for the applicability of magnetic bubble devices.

The second phase will address the low-level interface requirements for the MBB-80 Bubbl-Board (produced by PC/M Inc.) when interfacing with either the Intel 8080 or Intel 8086 microprocessor. The purpose of this phase will be to: (1) verify the operational characteristics of the MBB-80; and, (2) design and implement the low-level systems software necessary to interface the operating system's I/O structure with the magnetic bubble memory controller.

The third phase will address the issues necessary to implement the interface of the bubble memory system with the operating system's primitive secondary storage access routines. The tasks necessary in this phase are to: (1) design a memory organization and management scheme for the magnetic bubble memory; and, (2) design the interface such that the magnetic bubble memory <u>appears</u> as a "standard" mass storage device (disk) to the host operating system.

The fourth phase is the actual interface of the MBB-80 Bubbl-Boards into the CP/M-86 operating system. The

interfaces and designs developed in the second and third phases are applied in this phase. A generalized, table-driven, "basic input/output system" (BIOS) is developed which will allow the utilization of MBB-80 Bubbl-Boards (as "disks") by the CP/M-86 operating system along with conventional floppy and hard disks.

II. BACKGROUND OF BUBBLE MEMORIES

A. MAGNETIC BUBBLE DOMAINS

The entity known as the "magnetic bubble" has been much talked about in the context of solid state memory technologies. This section will present a description of what a magnetic bubble domain is and will describe some of its properties. No attempt will be made to present a comprehensive explanation of magnetic substances or magnetism, but rather the basic theories of magnetic domains will be put forth.

Certain elements and their alloys (Fe, Co, Ni, Gd and Dy) along with other substances exhibit the well-known property of magnetism or, more properly, ferromagnetism [Ref. 1: p. 619]. This property permits a material's atoms to achieve a high degree of alignment despite the atoms' tendency towards randomization due to thermal motions. Adjacent atoms interact and couple into rigid semi-parallel patterns. These patterns are known as ferromagnetic domain structures and are localized within a specimen. Materials can be cut such that their direction of magnetization is along a single axis (viz., along one particular direction) and are known as uniaxial ferromagnets.

Several important properties of ferromagnetism are exhibited when a magnetic substance is subjected to an applied (external) field. First, a relative increase in the external field of 0 to 0.01 will cause a relative increase in the substance's magnetic field of 0 to 1000 [Ref. 2: p. 2]. This factor of 100,000 occurs primarily in a long, thin sample or in a closed ring of some form. Secondly, if a single, thin, crystal sheet (film) of certain uniaxial ferromagnetic materials is cut perpendicular to the axis of natural magnetization (see Figure 2.1(a)), the domain structure is found to be one of wavy, or serpentine, strips having alternating directions of magnetization which are perpendicular to the surface of the sheet [Ref. 3: p. 86].

It is the combination of these two properties which supplies an environment for a magnetic bubble domain. A thin crystal film as described above, in the absence of an external field, will have a volume of serpentine strips magnetized in one direction which equals the volume of strips magnetized in the other direction, resulting in zero net magnetization. Upon the application of an external magnetic field perpendicular to the film, the strip domains magnetized in the direction of the field will increase in volume as the oppositely magnetized domains shrink in volume

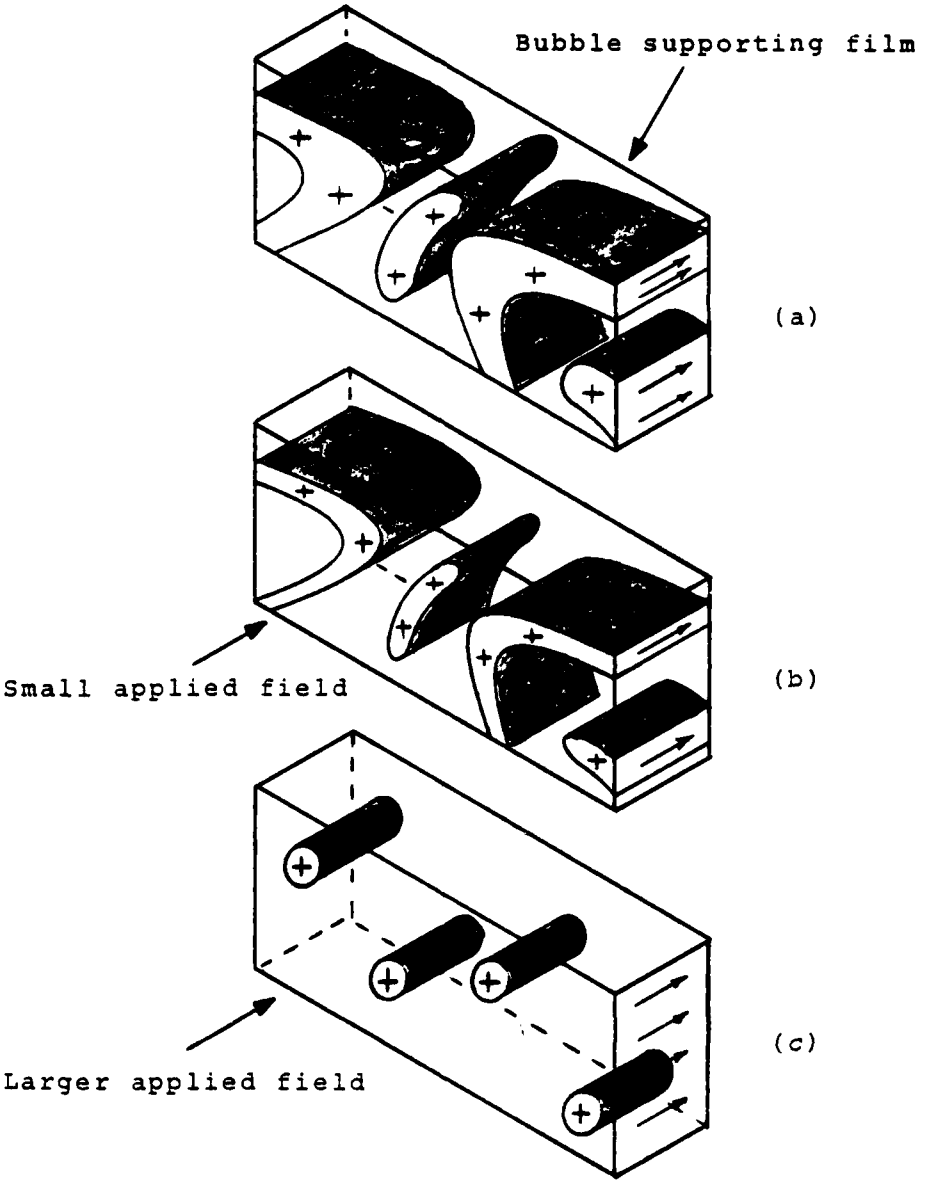


Figure 2.1 (a) Serpentine Strips, (b) Magnetized Strips, (c) Cylinders

[Ref. 3: p. 86]. This phenomenon is the result of the process of energy minimization and is shown in Figure 2.1(b). As the external field increases in strength, a field value will be reached at which the shrinking domains contract into circular cylinders; it is these cylinders which are known as "magnetic bubbles." These cylinders are shown in Figure 2.1(c). A further increase in the field will ultimately result in the total collapse of the shrinking domains, leaving the film saturated (viz., magnetized in one direction only) [Ref. 4: pp. 3-4].

The applied field, known as the bias field, is essential for the stability of the bubbles within a substance. The bias is typically on the order of 100-200 Oersteds (a unit used to measure magnetic strength), which can be easily provided by small, permanent magnets. This allows stable bubble existence independent of any power source, which is the foundation for non-volatile storage media. The bubble itself is maintained by a combination of three forces. The stable equilbrium of the domain is preserved by the magnetization of the bubble itself producing internal magnetic pressure which opposes the squeezing force of the applied field. The bubble domain maintains its circular shape because of the force of the magnetic surface tension of the wall which surrounds the domain. [Ref. 2: p. 10]

Clearly, the absence or presence of a magnetic bubble domain can be used to represent a zero (0) or a one (1) for data storage. However, there are several additional requirements which must be met before this technology can be considered for use as a data medium. One of these properties is the mobility of magnetic domains. A bubble will move towards any position which minimizes energy. Such locations can be defined and created by having small, reduced fields of external bias. Unbalanced forces acting on the wall of the bubble will cause the bubble to move in the direction of the reduced bias field. By laying out a "track" of permalloy (nickel-iron alloy) on the magnetic film and selectively altering the local bias on the track, it is possible to move bubbles along a prescribed path. It is important to note that, although this is similar to bits on a magnetic tape, there are no mechanical, moving parts involved as the bubbles move along this closed track. The fact that the bubble domains are only a few microns in diameter and may move at velocities in excess of several meters per second can provide data rates in excess of several megabits per second [Ref. 2: p. 10]. The remaining requirements of a storage medium will be presented in the next section. It will be seen that magnetic bubble domains can meet these requirements as well.

B. BUBBLE DOMAIN DEVICES

This section will discuss the basic operations necessary to support bubble domain devices. These operations include bubble propagation, bubble domain generation and bubble domain detection. Some basic bubble memory device organizations will be presented along with the theory and problems associated with these organizations.

The effect of a bias field on predefined tracks was explained as the basis for bubble domain propagation. These tracks are in fact analogous to conventional electrical transmission lines in that the track carries a signal (bubble) to various parts of the system. To meet the needs of data storage it is necessary to be able to "field access" the propagation track (viz., access a specific location). This implies multiple tracks (for more than one bit) on a bubble domain device that are all controlled and synchronized by one external magnetic field applied to the entire device. By rotating this field, known as the drive field, a magnetic wave can be caused to travel through the device. The bubble domains "ride" this magnetic wave and, thus, propagation takes place [Ref. 2: pp. 16-17]. Of course, it is necessary to be able to make the bubble domains change their direction of movement. Special

permalloy circuits have been designed to provide this function. Straight tracks in the form of "T-bar" circuits, combined with special 90 degree and 180 degree corners, form a basic storage array [Ref. 3: p. 87]. The "T" shape is used because of the magnetic field effects found around the long stem of the "T". Bubbles that move up this stem are trapped under the crossbar. As the drive field rotates, the bubble follows around the top of the "T", eventually moving perpendicular to its original direction (see Figure 2.2).

The operation of bubble domain generation involves the creation of bubbles (writing 1 bits) within the device. Most generation is done by a process called nucleation. A current of a few hundred milliamps, maintained for approximately 100 nanoseconds, is used to create a localized field in opposition to the bias field. This reverses the magnetization on the film, which causes the creation of a new bubble -- its size and position being finally stabilized by the bias field [Ref. 4: pp. 3-7]. It is noted that the process of nucleation is temperature sensitive and an implemented system must provide a means of varying the generation current to meet large temperature changes (failed nucleation or multiple nucleations can occur).

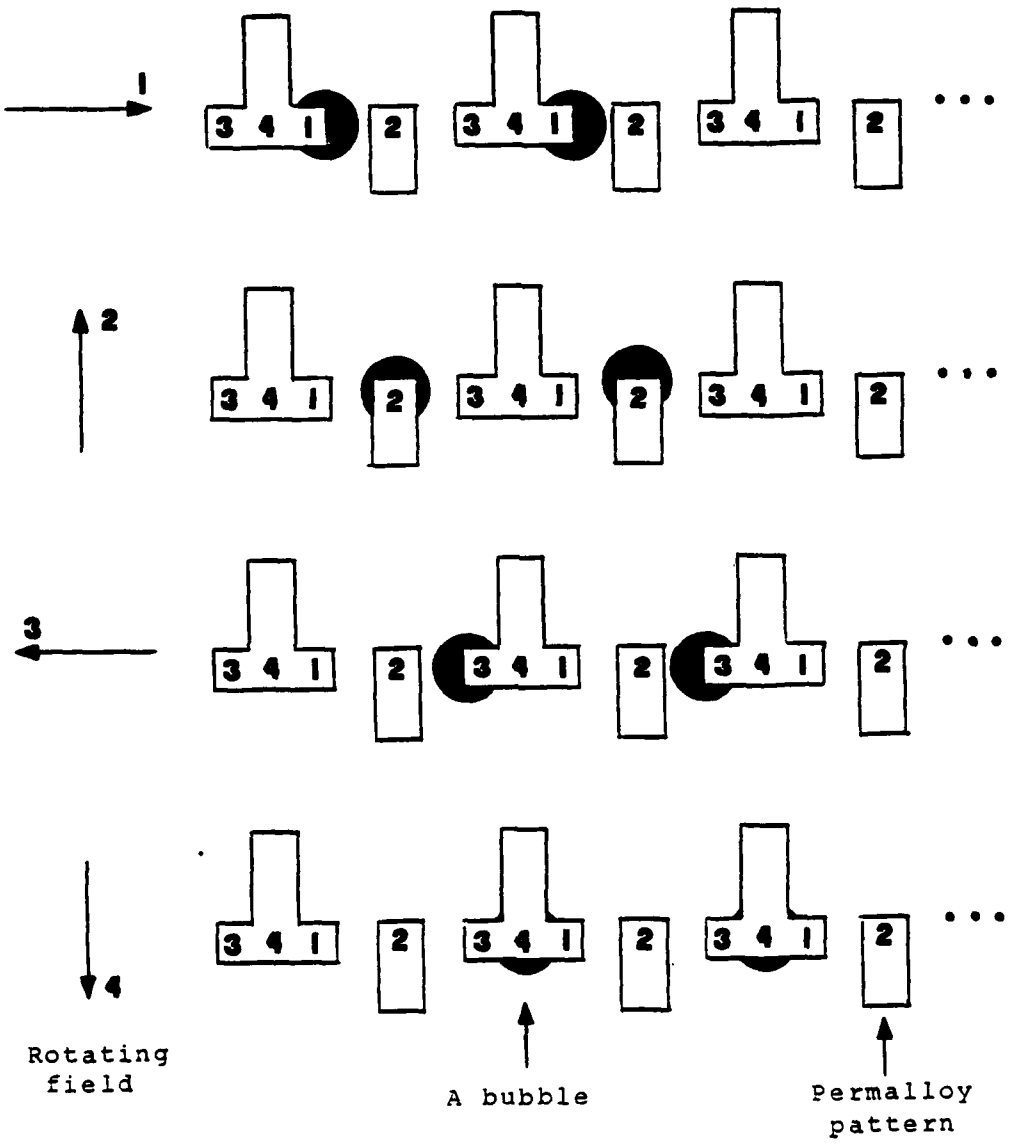


Figure 2.2 "T" Bar Movement

There are several approaches to the problem of bubble domain detection, or reading bits. One technique is a non-destructive readout scheme. A magnetic domain has associated with it a small magnetic field. As the bubble passes a suitable sense amplifier detector circuit, there will be a small change in the resistance of the circuit due to the magnetic field of the bubble. This detector is known as a magneto-resistive sensor and has the advantage of being a passive (no overhead) detection scheme. Unfortunately, the "signal" that is measured, or read, is but a fraction of the total power of the bubble domain. The second approach is one of a destructive readout. The bubble domain is side-tracked onto a special detection/generation track. Here the full power of the domain is sensed (causing the destruction of the bubble if one is present) for a stronger readout signal. The bubble (if present before readout) must now be re-generated and returned to the storage track [Ref. 5: p. 41]. This re-nucleation obviously requires more power and more supporting devices than the passive readout schemes.

The operations possible with magnetic bubble domains can result in a wide variety of architectures for bubble devices. Some of the more sophisticated designs will be

presented in Section D of this chapter. An explanation of the first, and simplest, bubble domain device will be discussed here.

An analysis of the magnetic device from a top-level view reveals a basic structure as seen in Figure 2.3. All devices will correspond to this structure and, by some means, implement the functional blocks as seen in this figure. Only the function of redundancy management was not discussed in the above sections. This is basically the issue of how manufacturing techniques result in a certain chip yield (viz., the <u>useable</u> portions of each bubble chip). It is sufficient to say that various mechanisms are available to provide redundant storage capability in a device and to keep a map of this redundancy. One method will be discussed in Chapter IV, Section A.

Magnetic bubble devices are serial storage devices with block access capabilities. They are similar to conventional electromechanical media, but with several major differences. Bubbles can be stopped and started at the bit level while most devices are block-oriented at a larger data volume. Bubbles do not have mechanical addressing aids like start-of-tape, disk tracks and sectors or optically-sensed index markers. Some other means of identifying and locating

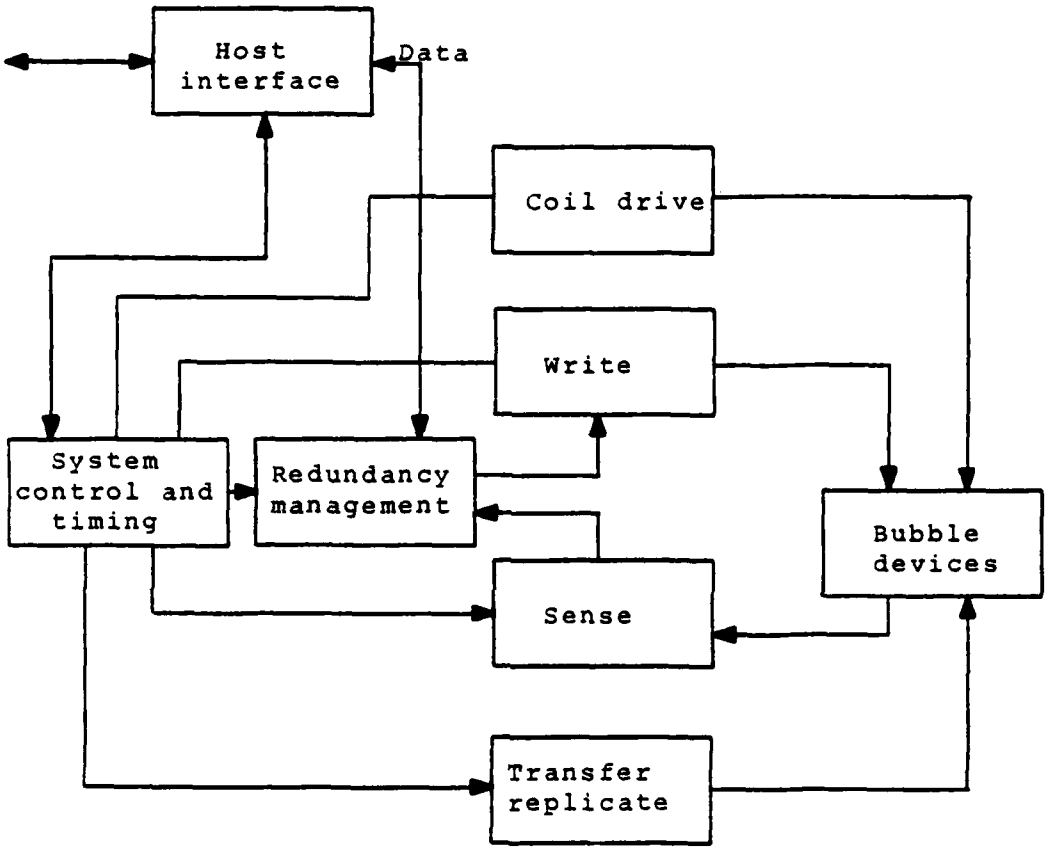


Figure 2.3 Basic Magnetic Device Functions

data is necessary. It is the chosen means of addressing that influences the device design of bubble storage.

The simplest magnetic bubble domain device uses the shift register organization. This is depicted in Figure 2.4(a). Bubble domains rotate around a fixed, closed loop with a simple generator and detector circuit. Average access times require propagation of a bubble through half the register. Transfer rates are dependent on serial bit-by-bit transfer through the detector. This simple device points out the three operational characteristics (which the shift register does not address efficiently) that influence the design of bubble devices: (1) need for high data density; (2) fast access time; and, (3) fast transfer rates.

The major/minor loop chip organization depicted in Figure 2.4(b) was the first attempt to address the need for improvement in these characteristics. This scheme is basically one of block transfer between the minor storage loops and the major operational loop. Bi-directional transfer gates allow a block of data equal (in bits) to the number of minor loops to be transferred to/from the major loop in a single operation. Transfer of all bits in parallel is achieved by a pulse to the common transfer bar

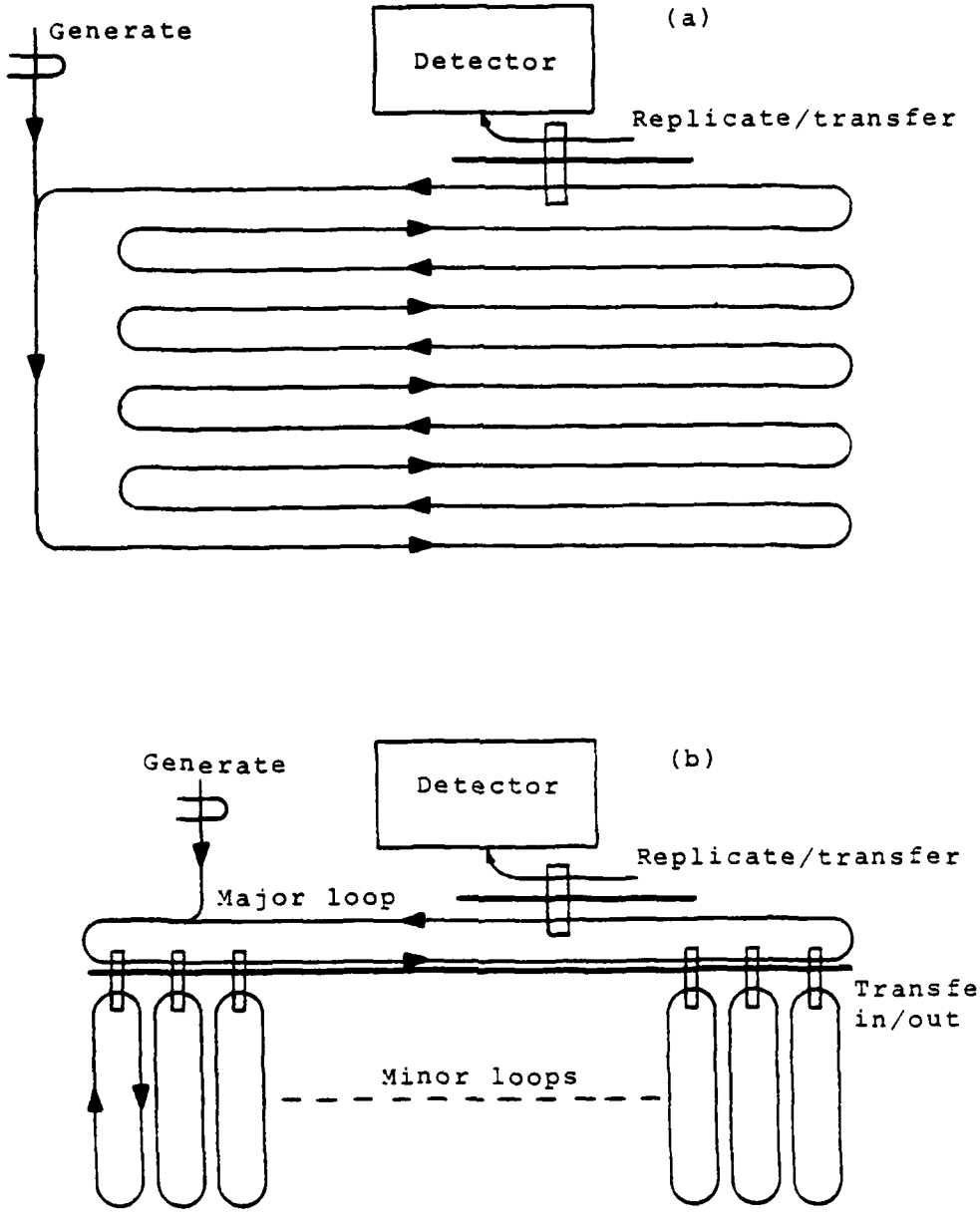


Figure 2.4 (a) Shift Register Architecture,
(b) Major/Minor Loop Architecture

between the major loop and the minor loops. The minor loops rotate in synchronization with the major loop. The major loop makes one revolution to perform its operation, then the data on the major loop is read back to, or written into, the minor loops. This clearly has the advantage of being a simple, easy-to-build device that provides some degree of increased data storage and access times. However, this device, implemented as a single entity, still suffers from serial readout and slow external transfer rates.

The next section will digress to discuss the history and development of bubble domain device technology. It is presented merely as a historical perspective to provide the context for the discussion of architecture and technology in Section D of this chapter.

C. HISTORY AND DEVELOPMENT

Bubble domain devices are a relatively new technology. The discovery of garnets, a glasslike substance, in 1956, allowed the fabrication of an environment condusive to magnetic domains. In 1959, the first bubble and serpentine domains where observed in certain ferromagnetic substances. A. H. Bobeck, of Bell Telephone Laboratories, presented the first description of bubble devices at the 1967 International Magnetics conference. Bubble domains were ignored at that time. [Ref. 6: p. 3]

The debut of the bubble domain occured in 1969, when Boteck, at the INTERMAG conference, updated his 1967 presentation. He clearly showed the feasibility of controlled bubble propagation in a shift-register device, along with bubble generation, replication and detection. For the first time, bubble domains were seen in the context of mass memory media. The technical interest generated at that conference soon had an effect on the business community.

Bell Systems, where the first bubble devices were designed, utilized this technology for repertory dialers, voice message recording and fixed-head-file replacement. Hitachi was the first company to announce a magnetic bubble memory product (Oct 1975) which was an 18-chip, 32K byte unit intended for office machines. Hewlett-Packard quickly followed with applications in desktop calculators.

Texas Instruments introduced the first general purpose bubble device in 1977. This is a 92K bit memory module which they utilized in their portable terminals. It is interesting to note that at this time several of the largest semiconductor memory manufacturers (Intel, Signetics, Rockwell International and National Semiconductor) entered the arena of bubble devices.

The early 1980's have brought the advent of 1M byte bubble devices with transfer rates in excess of 800 Kbits/sec. A detailed analysis and comparison of the different memory technologies and applications will be presented in Chapter III. The historical development of bubble memory devices can be referenced to the basic characteristics and operations presented in this chapter. The driving inpetus has been on providing denser packaging (more bits), faster access times and higher transfer rates. All of these factors have been necessarily constrained in the context of marketability and manufacturing costs. These considerations have produced many newcomers into the field along with revolutionary designs and architectures for magnetic bubble devices. However, the develcpment of a new technology that must simultaneously compete with established technologies (semiconductor, disk) has proven to be a limiting factor in the advancement of magnetic bubble devices (TI and National withdrew from the market in 1981 for reasons of profitability).

D. CURRENT TECHNOLOGY AND ARCHITECTURE

The attempt to improve the performance characteristics of bubble domain devices has proceeded along three distinct paths. First, has been the improvement of the components

making up the bubble device itself (viz., sense amplifiers, garnet substrates, etc.). Secondly, there has been much effort directed at finding an optimal architecture for the basic major/minor loop organization. Finally, the extensive use of support circuitry and sophisticated controllers is presenting a more simplified logical view (as seen externally) of magnetic bubble devices.

The design of physical components for the bubble devices is inherently coupled to the issues of magnetism, field electronics and garnet manufacture. An extensive discussion of these topics, however, is not within the scope of this thesis. Therefore, only mention of the areas of work in current research will be made here. The coil drivers, as originally described, produced a sine wave which propagated bubble domains throughout the device. These sine waves, which start and stop precisely, are difficult to implement at a low cost and have, therefore, been replaced by devices that generate triangular or trapezoidal wave forms [Ref. 5: p. 41]. Bubble detection, whether destructive or non-destructive, has non-trivial current requirements for the sense amplifiers. A reduction in the number of and power requirements for current sources is a primary goal of detection circuit design. Finally, the issue of high bit

density per unit cost, as in all memory devices, is being addressed by new garnet substrates. The work in this area has the goal of reducing the size of the bubble domains and putting as many tracks as possible on a chip while avoiding inter-bubble interference [Ref. 7: p. 63]. Current technology is supporting 1 Mbit devices with areas of less than one square centimeter and with a bubble domain diameter of two (2) microns.

The first bubble domain device architecture, the shift register, suffered from two main inadequacies: (1) a single defect in the shift register chain resulted in a bad chip; and, (2) data just entered had to be cycled through the entire shift register chain to be read, resulting in slow data access. The major/minor loop design addressed these problems. Data is generated in a major loop, circulated, read and rotated back to be restored in the original minor loop positions. Shorter cycle times are achieved if this need to restore data is removed. This idea was incorporated into the "block replicate" architecture. This is a multiloop arrangement where the minor loops communicate with a read track via replicate/transfer gates, allowing reading without disturbing the minor loop data (see Figure 2.5). Erasure is accomplished by activating transfer without

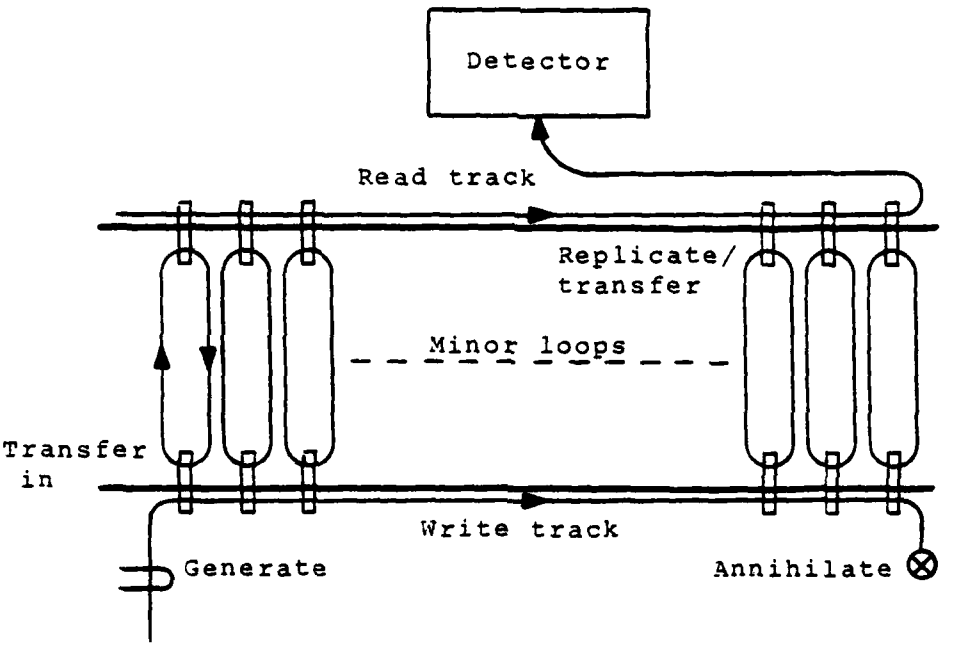


Figure 2.5 Block/Replicate Architecture

replicate. A separate write track allows block data to be written to the minor loops via transfer-only gates. The idea behind the replicate/transfer gate is that a bubble domain is replicated (by splitting or nucleating a new bubble) and then transferred to the read track for processing by the detector. The conventional major/minor loop design did this one bit at a time on the major loop whereas the block/replicate design replicates, in parallel, all the minor loop bits in a block.

The physical makeup of bubble domains and their resulting interactions requires that minor loops have bubble domains two (2) bits apart (viz., an empty position between every position where there could be a domain). Consequently, a major loop or read/write tracks could only generate on every other cycle, that is, they would cycle once uselessly while the minor loops cycled to bypass the empty positions on the major loop. Data can be read on every cycle by splitting the data storage into odd bits (loops) and even bits (loops) [Ref. 3: p. 95]. This architecture is depicted in Figure 2.6. To perform a write operation, the entire block is generated in both write tracks. The odd and even generate tracks are aligned simultaneously with the minor loops and the write takes

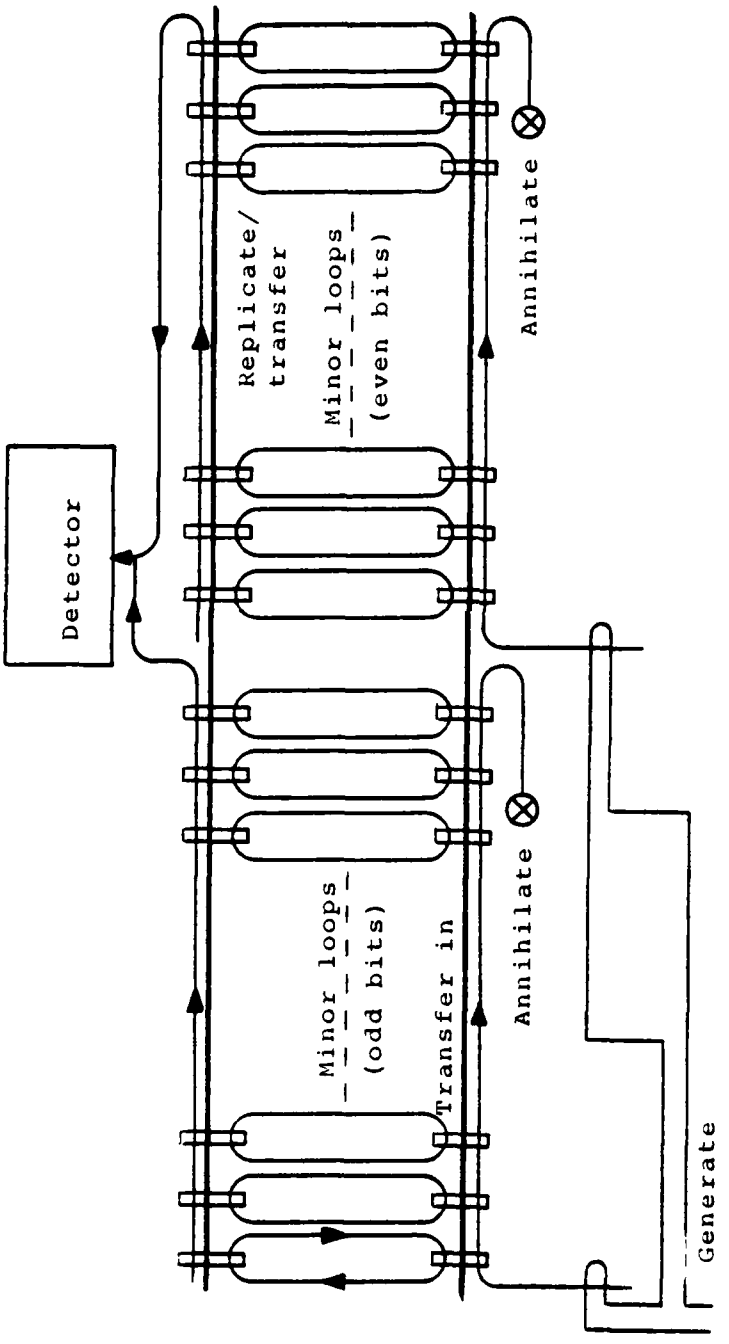


Figure 2.6 Block/Replicate Odd/Even Architecture

place. To perform a read operation, the replicated gates are activated on the odd and even storage loops. The two tracks are one bit apart so that the odd and even tracks are interlaced as they go to the detector, providing a read on every bit position.

All the multiloop architectures use redundancy to solve the problem of defects in chip manufacturing. Extra storage capacity is provided on the chip by having more minor loops than are actually required to meet the device memory capacity. Bad loops, normally discovered in factory testing, are located and put into some form of a map. Defective loop addresses are usually stored in a PROM within the bubble controller or in some of the redundant loops themselves. [Ref. 3: p. 87]

To become an economically practical and versatile device, it is essential that bubble memories present a functionally simple and logical view to potential users. Much effort has been put forth in the area of support circuitry which handles the low-level functions involved with the management of bubble devices. The biggest addition to the support circuitry has been in the area of bubble memory controllers. These controllers (which are usually 40-pin HMOS devices) provide bus interface, generate all

system timing and control, maintain memory address information and process the user's external software requests and commands to the bubble devices [Ref. 8: p. 57]. The conceptual purpose of the controller is to make the magnetic bubble memory look like a peripheral to the host computer. The sense amplifiers used for detection have been incorporated to include multi-channel capabilities (viz., to handle parallel readouts from more than one device to allow high data transfer rates). This results in a logical memory organization which can span "n" devices, where "n" is the number of bits in the host system's word size or data bus size. Data protection and save-circuitry have been provided to prevent bubble contamination in the event of a power loss, which can lead to a situation where loops are not rotated back to their starting point. This is necessary for correct addressing. The controller, utilizing a bad-loop map, also automatically substitutes redundant loops for bad loops on a chip.

The current architecture and technology of bubble domain devices are influenced by the need to compete with existing secondary memory devices. Consequently, much effort is being put into both the physical manufacturing of the bubble devices as well as into the logical architecture and user

interface. It is clear that any architecture must allow magnetic bubble memories tc be easily interfaced to existing computer systems.

The next chapter will provide an analysis and comparison of magnetic bubble devices to current memory technologies, with particular emphasis on the specific strengths and weaknesses of magnetic devices. Applications fcr magnetic devices will also be discussed in depth.

III. APPLICABILITY OF MAGNETIC BUBBLE MEMORIES

A. COMPARISON OF MASS STORAGE TECHNOLOGIES

Magnetic bubble memories should not be considered to be in direct competition with existing, well-established forms of non-volatile storage. Rather, bubble memories should be viewed as a secondary storage technology which can fill the well known capacity/cost and performance/cost gaps in conventional memory hierarchies.

In Figure 3.1 are plotted the areas inhabitable by a wide range of memory technologies. As can be seen in Figure 3.1, there is a large gap between core technology and fixed-head disk technology. At present, attempts to fill this gap are being made by electron-beam accessed memories (EBAM), charge-coupled devices (CCD) and magnetic bubble memories (MBM). Although EBAM probably has the lowest potential cost per bit of the three technologies, it requires fragile vacuum components which severely limit applications.

CCD technology has not sufficiently surpassed dynamic RAM technology to become preferable from either an economic or a performance standpoint. Currently, CCD memory access times (approximately 100 microseconds) are much slower than

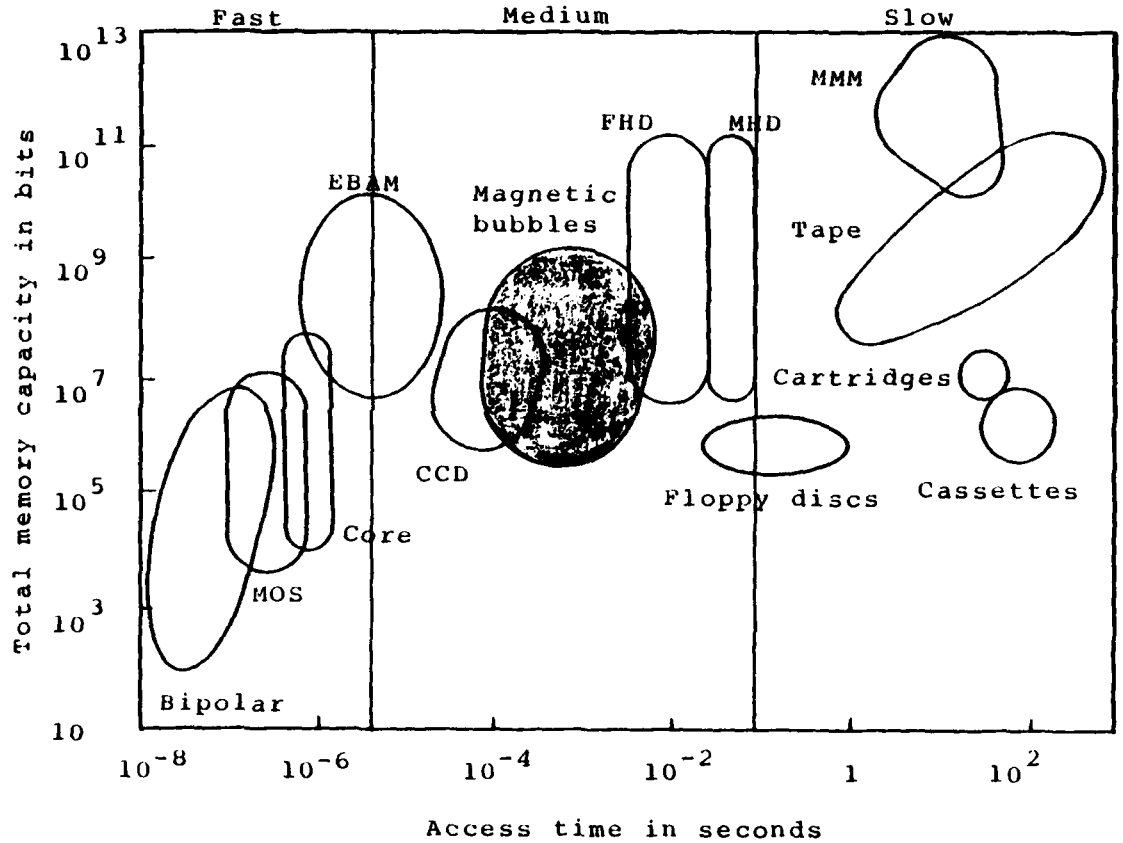


Figure 3.1 Memory Technology Access Times and Capacities

thcse of semiconductor RAM (70-2000 nanoseconds). An additional disadvantage of CCD memory is its susceptibility to alphaparticle radiaticn. As is the case with RAM technclogy, as memory densities have increased, the capacitance needed to store the charge for each bit has decreased, making it more probable that an alphaparticle strike will cause a soft error. [Ref. 9]

Magnetic bubble memories, cn the other hand, have the advantages of non-volatility, higher density and lower cost per bit over CCD and RAM technologies, and the advantage of solid-state technology over EBAM. Evaluaticn of the performance of magnetic bubble memories is usually accomplished utilizing the same parameters as those used for evaluation of floppy disk devices. Valid comparisons can be made between the performances of the twc technologies because of their common roles as secondary storage technologies.

Magnetic bubble memcries are organized as shift registers for block access, with the natural block size, referred to as a page, being equal to the number of minor locps. Access to data is accomplished by shifting bubbles in the minor loops and transferring the appropriate page to the major loop. The data is then read or written by

shifting bubbles around the major loop. This organization allows for the computation of both a seek time and an access time to parallel disk performance measures of the same names.

The seek time of disk systems is normally taken to mean the time it takes to move the read/write head to the track containing the desired data. This is analogous to rotating the minor loops in a magnetic bubble device to place the desired page on the major loop. Seek time for a bubble memory device is, therefore, dependent on the number of shifts required in the minor loops and the shift rate of the device. Current bubble memory architectures contain from 64 to 4096 pages in the minor loops and have a relatively common shift rate of 100 KHz [Ref. 10: p. 29]. Taking worst case to be a complete rotation of the minor loop at 10 microseconds per shift results in worst case seek times of 6.4 - 41.0 milliseconds. Assuming half of these values to be an average yields average seek times of 3.2 - 20.5 milliseconds.

Combining this seek time with the time required to rotate to the first bit of data in the read or write track yields the data access time for a magnetic bubble device. By assuming an average major loop size of 144 bits (the

actual major loop size of the TIB0203 92K bit device) and applying the shift rate of 100 KHz, a worst case read/write delay time of 1.44 milliseconds is obtained. Combining this delay with the previously computed seek time results in average access times of 3.92 - 21.72 milliseconds for magnetic bubble devices, which is considerably faster than the average access times of 115 - 500 milliseconds for floppy disk devices. [Ref. 11: p. 1]

The data transfer rate for a magnetic bubble memory is determined by the number of bits per page, the shift rate of the device and the number of cycles required to transfer the page of data out of or into the device. Basic transfer rates are 40 - 100 Kbits/second for individual magnetic bubble device organizations. These rates may be greatly improved by operating magnetic bubble devices in parallel (more than one device at a time). Bubbl-Tec's HDC/HDB-11 system, for example, utilizes four 1M bit bubble devices in parallel to attain a peak transfer rate of approximately 800 Kbits/second [Ref. 10: p. 29]. Such uses of parallel implementations allow magnetic bubble systems to achieve transfer rates in excess of those of floppy disk devices (125 - 500 Kbits/second).

The solid-state nature of magnetic bubble devices is a great contributing factor to their reliability. Since there are nc moving parts, the maintenance normally associated with electromechanical devices is avoided. An additional characteristic of magnetic bubble technology is very low error rates. Manufacturers' tests have produced hard error rates of 1 in 1 trillion bits and soft error rates of 1 in 1 billion bits [Ref. 11: p. 2]. A hard error occurs when a bit is read incorrectly during several consecutive read operations. Soft errors occur when a bit is read incorrectly cn one read cperation and correctly read on subsequent operations.

The final area of evaluation deals with the physical characteristics of the devices. Some additional properties attributable to the solid-state nature of magnetic bubble devices are low power requirements, light weight and ruggedness. Magnetic bubble memories may be sealed from the outside world and, thus, are immune to the effects of dust, humidity, dirt and vibration. Like mcst other technologies, however, magnetic bubble memcries do suffer temperature limitations. This limitation is due to the required matching of the temperature coefficient of the chip garnet to that of the permanent magnet. Currently, the specified

operating temperature range for most bubble devices is from 0 to 50 degrees Celsius but non-operating temperatures may range from -40 to +85 degrees Celsius without loss of data [Ref. 11: p. 2].

Magnetic bubble memory technology can provide a high density, low power, rugged, reliable and non-volatile data storage media. It is expected that the cost of bubble memory devices will continue to decrease and their density will continue to increase, making them an even more viable alternative mass storage technology [Ref. 12: p. 38].

B. APPLICATIONS OF MAGNETIC BUBBLE MEMORY

The variety of applications for magnetic bubble memories is steadily increasing. As system designers begin to take advantage of the properties of magnetic bubble memory devices, increasing numbers of bubble memories are being designed into systems, added on as back-up storage or used to replace other storage technologies. The variety of applications for magnetic bubble devices includes word processing, voice synthesis, portable terminals, communications, numerical machine tool controllers, aerospace and defense applications as well as others [Ref. 12: p. 38].

The high performance and low cost of magnetic bubble devices are the two major characteristics driving most of the applications. Current prices for bubble memories are roughly 100 millicents per bit with projected decreases to less than 30 millicents per bit in mid 1982 [Ref. 10: p. 26]. Access times of currently available bubble memories are approximately ten times faster than those of movable head disks and the data transfer rates of the two technologies are comparable [Ref. 13: p. 53]. Some magnetic bubble memory systems have, however, attained data rates of 96 Mbits/second and a system addressability of 4096M bits [Ref. 14: p. 141]. Another performance advantage is the simple addressing scheme which requires only an address and a read or write signal. It is estimated that a bubble memory controller would have 1/4 to 1/2 the complexity of an equivalent disk controller [Ref. 15: p. 37].

Another major contributing factor to the increase in applications of magnetic bubble devices has been the development of custom interface and support circuits. These integrated devices free the system designer from the need to become intimately familiar with the electrical and magnetic properties of bubble memories, thus, allowing more time to be spent on the system aspect of the application. There are

alsc many complete magnetic bubble memory system assemblies which can be plugged directly into DEC LSI-11s, Intel MULTIBUS systems, TI 9900s, S-100 systems and STD-bus machines [Ref. 10:p. 26]. Custom constructed systems require no separate chassis or power supply and can be ccnstructed entirely on printed circuit boards that can plug directly into existing bus structures.

Research conducted by IBM (San Jose, Califcrnia) has indicated that magnetic bubble memories must have a capacity of at least 4M bits in order to challenge RAM devices on the basis of cost. Bubble memory devices are approaching this density with 1M bit devices currently cn the market (TIB1000, Intel 7110 and National NBM2011). Rockwell has demcnstrated a 4M bit device developed under military contract and Bell Labs has fabricated an experimental 11.5M bit bubble device which is only 1.3 inches square. [Ref. 9]

Since magnetic bubble memories are of a solid-state, non-volatile technology, they are ideally suited for pcrtable applications as well as for providing additional stcrage for traditional and parallel processing systems. The compactness, low power requirement, quietness and low maintenance requirement have made bubble devices ideal for office equipment applicaticns. Additionally, the ruggedness

of the devices, when combined with the above characteristics, makes them ideal for use in the harsh environments often encountered in control and military applications.

IV. DESCRIPTION OF THE DEVELOPMENTAL SYSTEM

A. TIB0203 MAGNETIC BUBBLE MEMORY

The TIB0203 magnetic-bubble memory is a non-volatile, 92,304 bit, bubble memory chip. The chip is manufactured as a 14-pin dual-in-line package which contains the coils for prcviding a rotating magnetic field, a permanent magnet to maintain data storage and a magnetic shield structure. The TIB0203 is designed as a conventicnal major/minor loop architecture with 144 minor loops (circular shift registers) of 641 bits each. Transfers of data to or from the single majcr loop are done in parallel. The major loop contains the detector circuits as well as the generate, replicate, and annihilate control functicns. [Ref. 16: p. 11]

Detection is accomplished in a passive scheme utilizing twc magneto-resistive elements. Th e elements are out of phase with each other and cperate on alternate cycles (viz., alternately reading bit positions). Noise produced in the circuit due to circuit laycut, control pulses and from the magnetic fields is reduced by cancellation when the elements are used with a bridge circuit and an external differential amplifier. [Ref. 16: p. 14]

Generation of bubble domains is done via nucleation as a specified current pulse is sent through the generate loop. Transfer-in is accomplished as follows: (1) a data string equal in length to the number of minor loops (called a page) is generated; (2) this string is shifted such that the first bit is positioned over the first minor loop; (3) the transfer gates are energized. Each of the 641 minor loop page positions is useable. Transfer-out is accomplished in the reverse manner. Once a page is on the major loop it is eligible for one of two operations in a serial bit-by-bit manner: replicate or annihilate. [Ref. 16: p. 11]

A replicate operation causes the bubble domain to be stretched, then split in two with one bubble diverted to the detector and the other diverted back to the major loop and subsequently to the minor loop for storage. This procedure provides for a non-destructive readout. Annihilation is provided by transfering the bubble domain off the major loop and into the detector track where it is propagated off the chip.

The chip is manufactured with 157 minor loops, which provides a redundancy of 13 minor loops. Defective minor loops are identified at the factory and a map is printed on the device before shipment. The map has the addresses of

defective loops printed in hexadecimal and it is the responsibility of the controller to prevent the use of these bad loops. [Ref. 16: p. 12]

The coil drive for the TIB0203 uses triangular wave forms generated from two orthogonal coils that are driven 90 degrees out of phase. A cycle is the time required for the magnetic field to rotate 360 degrees. Minor loops are spaced two bits apart with one bit separation on the major loop. Therefore, all major loop operations are performed at half the drive frequency. The drive frequency for the TIB0203 is 100 KHz. [Ref. 16: pp. 13-14]

The TIB0203's components and specifications are completely described in Reference 16, the "TIB0203 Magnetic-Bubble Memory and Associated Circuits Manual." Operating characteristics, block diagrams and environmental conditions for the function timing generator, sense amplifier, function driver, coil driver and thermistor are also included in this manual.

B. PC/M MBB-80 BUBBLE MEMORY SYSTEM

MBB-80 Bubbl-Board is the registered trademark of a magnetic bubble device marketed by Bubbl-Tec, a division of Pacific Cyber/Metrixs, Inc., located in Santa Clara, Califcrnia. The MBB-80 is a complete bubble memory storage

system designed to be compatible with all 8-bit and 16-bit microcomputers that utilize Intel's MULTIBUS architecture. The board provides 92,304 eight-bit bytes of non-volatile memory as well as all required control logic and buffering necessary to interface to the MULTIBUS system.

The entire system is contained on one multi-layer, printed-circuit board. The printed-circuit board has the standard MULTIBUS dimensions and requires one card-cage slot on the MULTIBUS. The board is built around eight (8) of the TIE0203 bubble memory devices described in the preceding section. All necessary support chips are included on the single board. The functions of the controller are provided in hardware and include the following primitive commands:

Fill Buffer	Read Multiple Pages
Empty Buffer	Initialize
Write Single Page	Read Status
Read Single Page	Enable/Disable Interrupt
Write Multiple Pages	Reset

Host interface with the controller is via memory-mapped I/O, using sixteen (16) consecutive user-defined locations in the CPU address space. The MBB controller can be set to recognize any sixteen consecutive addresses on a 16-line or 20-line address bus. These sixteen addresses correspond to sixteen registers in the bubble memory controller which are utilized to read status information, set MBB-80 board configurations and perform read/write operations.

The MBB-80 typically consumes less than 20 watts of power. Voltage requirements consist of +5 volts at 1.5 amperes, +12 volts at 200 milliamps and -12 volts at 700 milliamps. Logic is provided to protect stored data during power-up, power-down and when unexpected power failures occur. The MBB-80 can operate in a temperature range of 0 to 50 degrees Celsius. The magnetic environment is less than 20 Oersted at the bubble device and the board weighs 18 ounces. A complete description of the MBB-80, its printed-circuit board layout and schematic diagrams are contained in Reference 17.

C. DEVELOPMENTAL SYSTEM

The INTELLEC Double Density Microcomputer Development System (INTELLEC DD MDS) with an iSBC 86/12A single-board computer, an iSBC 202 double density disk controller and the CP/M-86 (version 1.0 as modified by Reference 18) operating system (hereafter referred to as CP/M-86) is the host system for this implementation. This system is located in the Microcomputer Laboratory at the Naval Postgraduate School, Monterey, California, and will be described in greater detail in the next section. This host system was found to have a severe inadequacy in the area of software development tools. The current CP/M-86 operating system had no

interface to a printer. The CP/M-86 resident text editor (ED) consists of relatively primitive commands which do not allow a wide range of text manipulation. For these reasons an alternative system had to be chosen for use in software development.

The text editor chosen was the screen-oriented editor of the Altos UCSD Pascal (Version 1.4b) system. Required Intel 8080 and Intel 8086 assembly language programs were written in files created utilizing the Pascal system editor. The overall efficiency of software development was greatly enhanced by the use of this editor. Once a file was completed, it was transferred to the Altos CP/M-80 (Version 2.2) system by executing the 8080 assembly language program, CPXFER, which executes under CP/M-80 (hereafter referred to as CP/M). CPXFER is a Naval Postgraduate School (NPS) Microcomputer Laboratory utility program that provides for the intersystem transfer of formatted files between the Altos CP/M and Pascal operating systems.

Once transferred to the CP/M system, Intel 8080 and 8086 assembly language programs could be assembled utilizing the standard, CP/M resident, Intel 8080 assembler (ASM) or Intel 8086 cross-assembler (ASM86), respectively. Errors encountered during assembly could be corrected utilizing the

CP/M resident editor (TED) and a corrected copy of the file transferred back to the Pascal system for purposes of consistency. Once a program is successfully assembled it is ready to be transferred to the INTELLEC DD MDS for execution.

The Intel 8080 or 8086 executable files (.CCM or .CMD respectively) are transferred to the INTELLEC DD MDS by utilizing the NPS Microcomputer Laboratory utility program called SDXFER for intersystem transfer of files between the single density INTELLEC MDS and the INTELLEC DD MDS. Files can be transferred directly from any CP/M compatible disk, on either drive of the single density MDS, to any CP/M compatible disk on either drive of the double density MDS, utilizing SDXFER.

All complete assembly language programs are maintained on the Altos UCSD Pascal system disks only. The Altos CP/M, double density MDS CP/M and double density MDS CP/M-86 system disks contain only executable files.

D. IMPLEMENTATION HOST SYSTEM

The final implementation utilizes the previously mentioned host system consisting of an INTELLEC Double Density MDS system and iSBC 202 disk controller, both under the control of an iSBC 86/12A single-board computer, and the

CP/M-86 operating system. Initial low level bubble memory testing was conducted utilizing the INTELLEC DD MDS and its resident Intel 8080 microprocessor. After initial testing of the device, all remaining development, testing and implementation utilized the iSBC 86/12A and its Intel 8086 microprocessor instead of the Intel 8080.

The INTELLEC DD MDS is a coordinated, complete computer system designed around the Intel 8080 microprocessor. The standard INTELLEC DD MDS system consists of an Intel 8080 microprocessor, two (2) 32K byte RAM memory modules, a monitor program with six (6) fully implemented I/O interfaces and a front panel control module, used to provide a 256 byte bootstrap program, the eight (8) level bus access control circuitry and a real time clock. These system modules are contained in an eighteen (18) card chassis which features the Intel MULTIBUS, which supports multi-processor configurations and allows for "master-slave" relationships between modules. The one addition to the standard system is the use of an iSBC 202 double density disk controller module to handle the dual floppy disk drives. [Ref. 19]

As previously mentioned, once past the initial testing phase, the INTELLEC DD MDS system was operated with the iSBC 86/12A. This was accomplished by removing the two memory

boards and the Intel 8080 CPU board and placing the iSBC 86/12A in a bus-master slot (an odd numbered slot) in the INTELLEC DD MDS chassis. The iSBC 86/12A is a single-board microcomputer based on the Intel 8086 16-bit microprocessor. Included on the board are 64K bytes of dynamic RAM, three programmable parallel I/O ports, programmable timers, priority interrupt control, serial communications interface and MULTIBUS interface control logic. [Ref. 20]

The CP/M-86 operating system utilized with the host system is a product of Digital Research. The specific operating system used was Version 1.0 with the modifications made in Reference 18. CP/M-86 is a microcomputer operating system for Intel 8086 based microcomputers. CP/M-80, the predecessor of CP/M-86, was designed for Intel 8080 based microcomputers and, as nearly as possible, file compatibility between CP/M-80 and CP/M-86 has been maintained. CP/M-86 provides built-in utility commands and transient system programs. Additionally, the user has the ability to execute user-defined transient programs. The system transient programs include a dynamic debugger (DDT86), a primitive text editor (ED) and an Intel compatible assembler (ASM86). [Ref. 18]

The entire implementation host system is located in the Microcomputer Laboratory at the Naval Postgraduate School, Monterey, California. Each of the individual components of the system (INTELLEC DD MDS, iSBC 86/12A and CP/M-86) is described in great detail in the reference listed after the discussion of the component.

V. LOW-LEVEL BUBBLE DEVICE INTERFACE

A. INTEL 8080 IMPLEMENTATION

Prior to interfacing the MBB-80 Bubbl-Board with the iSBC 86/12A, initial testing was conducted by interfacing the MBB-80 with the standard INTELLEC DD MDS system and its resident Intel 8080. The Intel 8080 was chosen for initial MBB-80 testing because of the authors' familiarity with Intel 8080 assembly language and because of the availability and utility of the existing CP/M-80 operating system and support programs (viz., DDT and TED).

Before any software interfacing or testing could be attempted, the hardware interface between the MBB-80 Bubbl-Board and the INTELLEC DD MDS system had to be constructed and verified. This interfacing required the modification of power circuits within the MDS system and necessitated the addition of a manual power-protect switch. The modification of power circuits was required to provide the 0.550 amps at -12 volts required by the MBB-80 Bubbl-Board circuitry. The remaining power requirements of the MBB-80, 1.0 amps at +5 volts and 0.12 amps at +12 volts, are available on the standard MDS system's bus. The manual power-protect switch was provided on an additional

development board and was required to protect the bubble devices during normal power-up and power-down. Bubble device contamination, as described in Reference 17, can result if the bubble devices are accessed while the power supplies are not within the specified tolerance of plus or minus 3 percent. The manual switch provides protection only during normal power-up and power-down. A more comprehensive power-protect system will be needed to provide full protection against inadvertant power loss in a production system. [Ref. 17]

Software interfacing and testing of the MBB-80 was conducted by writing and executing an Intel 8080 assembly language program called DIAG80.ASM (a program listing of DIAG80.ASM is contained in Appendix A). This program utilizes sixteen (16) consecutive addresses, beginning at a program defined bubble memory controller base of 04000H, as registers for communication with the MBB-80. The Inhibit ROM/RAM signals provided by the bubble memory controller allow the placement of the controller base address and the sixteen registers anywhere in the on-board 64K bytes of RAM not in direct conflict with CP/M-80 usage.

Initial attempts at execution of DIAG80 resulted in premature program termination. Attempts at debugging the

program by using DDT failed because single-stepping through the program resulted in proper execution. Full-speed execution, however, continued to result in premature termination at unpredictable and unrelated points in the program, indicating either a timing or a device compatibility problem. Further investigation revealed that the termination of execution was accompanied by a bus timeout signal from the MDS system (the bus timeout signal is initiated when a bus request is made and no acknowledgment signal is received within a specified time interval).

Monitoring various signals with an oscilloscope led to the detection of an inconsistency between the monitored signals and the specifications on the MBB-80 circuit diagram provided in Reference 17. While checking the comparators (utilized to determine if an address on the bus is that of a bubble memory controller register), it was determined that a signal of some sort was present on pin 7 of each of the three comparators. The circuit diagram indicated that these pins should all be connected to the common board ground. Upon contacting the designers of the MBB-80, it was learned that the circuit diagram currently being distributed was for Version B of the MBB-80. The correct circuit diagram, for Version D, was acquired and testing resumed.

During subsequent calls to Pacific Cyber/Metrixs personnel to confirm or question findings, it was learned that some special-purpose circuitry was connected to the comparators. This circuitry had been included for a special application design of the MBB-80 and was incorporated onto all boards currently being distributed. We were given the assurance of MBB-80 design personnel that this circuitry was in no way affecting the operation of our Bubbl-Board and that we could verify this by "grounding" pin 7 of all of the comparators. Temporary "grounding straps" were placed on all of the comparators to see if there was any affect on the operation of the MBB-80. Subsequent attempts at executing DIAG80 were all successful. Pacific Cyber/Metrixs personnel were informed of our findings. As a result, the designers of the MBB-80 are currently considering the inclusion of a manual switch on future MBB-80 boards to allow the user to select or bypass the special-purpose circuitry.

With DIAG80.ASM executing properly, initial testing of the MBB-80 was continued. Information was written into and read from pages of each device to verify that the bubble devices were error free. Additionally, information was written into the devices and power removed from the MBB-80. The MBB-80 was left for a 24-hour period and then data

retention was verified in each bubble device by reading back the previously stored information. Operation of the MBB-80 was satisfactory and the low-level read, write, controller initialization and device initialization routines had been verified to function correctly.

With initial MBB-80 interfacing and testing successfully completed and the low-level routines verified, advanced implementation and testing with the iSBC 86/12A was begun. The low-level routines were available for direct translation into Intel 8086 assembly language and the DIAG80.ASM program available as a model for future program construction.

B. USE OF THE CP/M-80 MBB-80 DIAGNOSTIC PROGRAM

The CP/M-80 diagnostic program, DIAG80.ASM, was designed and written for the purpose of testing the hardware interface between the MBB-80 and the INTELLEC DD MDS system. This program provides low-level routines which allow the user to verify correct write and read operations to and from the MBB-80. Although not originally intended to serve as such, DIAG80 can also serve as a low-level debugging tool to aid in systems program development.

DIAG80 is executed by executing the DIAG80.COM file located on the CP/M-80 system disk. Execution will cause the MBB-80 controller and all eight (8) magnetic bubble

devices to be initialized in accordance with Reference 17. The MBB-80 controller base (defined in DIAG80 by a constant) must be set to 04000H utilizing the address selection switches on the MBB-80. The program will then, at the discretion of the user, cause an eighteen (18) byte page to be either written into or read from one of the eight (8) magnetic bubble devices.

The user has the option of entering an "R" for a read, a "Q" tc quit or a "W" or any other character for a write. If the user-specified operation is to read a page, the user will be prompted for the single-digit bubble device number (0-7H) and the three-digit page number (000-280H) of the page to be read. The contents cf the specified page will be printed to the CRT along with the contents of the status register. If the specified operation is to write an eighteen (18) byte page, the user will be prompted for the twc-digit hexadecimal value to be written in addition to the bubble device and page number of the destinatcn. The twc-digit value given by the user will then be written into all eighteen (18) bytes of the specified page. If the user types a "Q", to quit, then the program terminates and a return is made to the CP/M operating system. No error checks are made to verify correct entries by the user. If

input values are outside the specified ranges the program will not function reliably.

C. INTEL 8086 INTERFACE CONSIDERATIONS

The actual interface and implementation of the bubble memory system were accomplished utilizing CP/M-86 and the iSEC 86/12A single-board computer. Several local modifications had to be made to the standard Intel iSBC 86/12A distribution board. The following description is provided to allow the verification of a correct board configuration when either duplicating this thesis work or continuing research on this system.

The address select pins for the iSBC 86/12A were configured to place the computer's on-board RAM in the lowest 64K byte segment. Therefore, address select switches one (1) and eight (8) are "on"; all others are "off". The following pairs of pins were connected together (jumpered) to provide the necessary interface to the locally modified Intellec DD MDS system: 3-4, 5-6, 68-76, 79-83, 87-89, 92-93, 127-128 and 143-144. The above iSBC 86/12A modifications are necessary for the correct operation of the iSEC 86/12A within the Intellec DD MDS system and are not necessitated by MBB-80 Bubbl-Board requirements.

The memory acquisition circuitry of the iSBC 86/12A will reference RAM on the iSBC 86/12A board for addresses 0-64K and onboard EPROM for addresses 0FFC00-0FFFFF (hexadecimal). Any memory reference outside these two ranges will activate the MULTIBUS acquisition circuitry. Consequently, bus override commands, or inhibit signals, issued over the MULTIBUS within the first 64K byte segment will have no affect on the iSBC 86/12A's RAM. This requires that the MBB-80's controller base be placed at an address outside of the first 64K bytes. Since the MBB-80 controller utilizes memory-mapped I/O to sixteen (16) consecutive memory locations, any 16 addresses that can be inhibited, will suffice. It was decided to provide the user with the ability to specify a segment base address for the MBB-80 controller in all of the CP/M-86 diagnostic (low-level interface) programs. Since the MBB-80 can decode 20 address lines, the controller's base address space can be placed anywhere within the 1M byte address space that isn't occupied by RAM or EPROM (which cannot be inhibited). The address specified to these programs must correspond to the address set on the MBB-80 address select switch.

In addition to the MBB-80 controller memory address assignment, the interrupt structure also has an affect on

the iSBC 86/12A configuration. The MBB-80 has two modes of operation: single-page mode and multi-page mode. The single-page mode, which requires no interrupts and was implemented successfully on the Intel 8080, also poses no problem for the Intel 8086. The multi-page mode, however, requires that specific timing requirements be met by the host computer in communicating with the MBB-80 controller. During transfers of data, the host must respond to the interrupts generated by the MBB-80 every 160 microseconds (signalling a completed transfer of one byte in a multi-byte transfer). These interrupts can be either generated over the MULTIBUS as "hard" interrupts to the iSBC 86/12A or the iSBC 86/12A can "poll" (read) the status register that is within the address space of the MBB-80 controller. A detailed description of single-page mode, multi-page mode and the required interrupts is given in Reference 17.

It was decided that the Intel 8086 implementation would be accomplished in steps. First, a simple, single-page mode program would be written utilizing the algorithms that were tested in the Intel 8080 implementation. Since the multi-page mode provides approximately four (4) times the effective transfer rate of single-page mode (45 Kbits/sec versus 11 Kbits/sec), it was deemed essential to utilize the

multi-page mode of operation in the final operating system interface. This required a decision on the method of detecting and servicing interrupts, which led to the development of a multi-page mode program that could operate in the "polling" mode or use interrupts generated over the MULTIBUS. To handle interrupts over the MULTIBUS, an additional modification was made to the iSBC 86/12A board: pins 72 and 80 were jumpered to allow IR1 (interrupt one) on the MULTIBUS to be processed as interrupt type 16 within the iSBC 86/12A microcomputer via the on-board i8259 programmable interrupt controller (PIC). It was also necessary to connect the IR1 interrupt on the MBB-80 board itself, as described on page 2-3 of Reference 17, which causes MBB-80 generated interrupts to be sent over the MULTIBUS on IR1. Along with the modifications to the Intellec DD MDS power supply and to the MBB-80 board detailed in Section A of this chapter, all hardware interface requirements have now been described.

D. INTEL 8086 IMPLEMENTATION

The implementation of the MBB-80 Bubbl-Board with the Intel 8086 was divided into two phases, with each phase having specific goals. The first phase was the implementation of a program which uses the single-page mode

of operation on the MBB-80, where the basic routines developed in the 8080 implementation would be utilized. The goal of this phase was to verify the successful operation of the MBB-80 with the iSBC 86/12A hardware using the CP/M-86 operating system. The second phase involved the implementation of a program which uses the multi-page mode of operation utilizing either the polling mode or interrupts generated over the MULTIBUS. The goals of this phase were: (1) verify that the multi-page mode of operation works; (2) determine which interrupt method is most desirable; and, (3) prepare and test software routines that can be utilized in the final operating system interface.

The single-page mode program, hereafter referred to as DIAG86S, was designed as a complete Intel 8086 assembly language diagnostic program for the MBB-80, requiring little operator intervention (as opposed to DIAG80.ASM -- the 8080 version). The program will continuously test every byte in each magnetic bubble device, recording all errors, until execution is terminated by the user. Three basic functions were to be tested: (1) initializing the MBB-80; (2) reading from the MBB-80; and, (3) writing to the MBB-80.

The algorithms developed in DIAG80 for initializing the MBB-80 controller and for reading and writing a physical

bubble page (18 bytes) were not logically altered. A direct translation of these routines was made from 8080 assembly language to 8086 assembly language.

It was considered desirable to utilize the Intel 8086's segmentation features to allow the future use of the full 1M byte address space available in the processor. Consequently, the simple "8080 memory model" was rejected in favor of the "compact memory model" which utilizes multiple, user-controlled segments (see Reference 21, pages 7-9, for a complete description of these models). Code segments (CS) and data segments (DS) are used only for code and data respectively, while the extra segment (ES) is used to address the MBB-80 controller ports at a user-defined base address (see Reference 22 for a description of ASM86 and segments).

DIAG86S was written and tested. During debugging, routine code and logic errors were encountered but no problems relevant to this specific implementation were discovered. Execution of this program on the iSBC 86/12A, under the CP/M-86 operating system, achieved all of the stated goals for this phase of the 8086 implementation. A complete listing of DIAG86S.A86 is contained in Appendix B.

The multi-page mode program, hereafter referred to as DIAG86M, is a diagnostic program that performs the same functional diagnostic tests as DIAG86S. In meeting the stated goals of this phase in the Intel 8086 implementation, several important issues were addressed. First, the programming of suitable interrupt handling mechanisms to service both MULTIBUS and polled interrupts from the MBB-80 was necessary. Second, a method for evaluating the desirability of these methods was needed. Finally, the routines that performed specific bubble memory functions had to be in a form suitable for direct application in the next step of this thesis, the implementation of the interface to the CP/M-86 operating system.

The two methods of handling interrupts are provided by a conditional assembly variable in DIAG86M. The boolean status of this variable (documented in the code) determines whether code is generated for a MULTIBUS interrupt or for the polled mode of operation. For the MULTIBUS interrupt (in addition to the above mentioned hardware modifications) three steps are required: (1) set up the interrupt vector in CP/M-86 low memory to handle the IR1 signal from the MULTIBUS; (2) program a trap handler at this interrupt vector; and, (3) programming the i8259 PIC to recognize and

properly interpret the interrupt coming in over IR1. A simple semaphore, set by the trap handler and interrogated by the bubble routines, is utilized to signify the occurrence of an interrupt from the MBB-80. The use of the polled mode merely requires the interrogation of the interrupt flag register at port offset 0FH in the bubble memory controller.

Both the interrupt mode and the polled mode were successfully implemented. Execution times for complete diagnostic runs were 47 seconds for both methods (timed with a conventional stopwatch). Due to the extra code and hardware modifications required for vector initialization, the decision was made to utilize the polled mode in the CP/M-86 operating system interface. Although this approach limits a future application with multiple processes requiring priority interrupts, this approach is consistent with the polled interrupt structure utilized by disk systems that are generated and distributed with the CP/M-86 operating system by Digital Research. It should be noted that the code and hardware modifications for the use of interrupt vectors included in this chapter are completely functional for future applications that require a prioritized interrupt structure using the MBB-80.

The bubble memory initialization routine used in DIAG86M is in the same form as that used in DIAG86S. However, the read and write routines used in DIAG80 and DIAG86S are based on using a physical, magnetic bubble memory, page number as an addressable unit for each transfer. Therefore, the foundation for the memory organization of the MBB-80 was developed which would be compatible with that expected by a CP/M disk structure. DIAG86M views the transfer as that of a logical CP/M sector of 128 bytes. Since a physical bubble page is 18 bytes and 128 is not an even multiple of 18, the last sixteen bytes of each logical bubble "sector" (144 bytes) will be ignored (wasted). A logical CP/M sector consists of 8 bubble pages of which the last 16 bytes on the last page of a bubble "sector" are not used. There are 640 bubble pages per device (chip), so there are 80 logical CP/M sectors (as well as 80 bubble "sectors") on each bubble device. The access of data on the Bubbl-Board now requires only a device number (0-7) and a "sector" number (1-80) on that device. A routine to convert a "sector" number to a starting page number of an eight page "block" was written and tested. This routine takes into account the fact that the multi-page mode requires a "skew" factor of 322 on each consecutive bubble page access. This skew factor allows the

rapid access of pages without making complete shifts of the major loops in the magnetic bubble devices. Mathematically, the starting page number is computed as follows:

SPN = ((SN-1) * 12) mod 641

where SPN = starting page number (0-640)
SN = MBB-80 "sector" number (1-78)
mod = modulo division (remainder)

A complete description of this "skewing" operation and the necessary programming considerations is provided on page 3-13 of Reference 17.

DIAG86M was written, tested and debugged in both the interrupt mode and the polled mode of operation. Execution of this program on the iSBC 86/12A, under the CP/M-86 operating system, achieved all of the stated goals for this phase of the implementation. A complete program listing of DIAG86M.A86 is found in Appendix C.

E. USE OF CP/M-86 MBB-80 DIAGNOSTIC PROGRAMS

DIAG86S.A86 is a single-page mode, 8086 assembly language diagnostic program for the MBB-80. Its purpose was to verify the correct operation of the MBB-80 under CP/M-86 but it can be used as a functional diagnostic program. Since it operates in single-page mode, no supporting interrupt structure is necessary for execution of this program.

This diagnostic is invoked by executing the DIAG86S.CMD file on the CP/M-86 system disk. The program will print appropriate messages and then request that the user key in a four (4) digit, segment base address for the MBB-80 controller. Only four digits can be keyed in, followed by a carriage return. Keying in more than or less than four digits, or invalid hex digits (viz., not in the range 0-F), will cause the printing of an error message and the user will then be asked to re-enter the segment base address. This segment base address consists of the high order 16 bits of the 20-bit address that is physically set on the MBB-80's address select pins. The address keyed in must match the MBB-80's address and the MBB-80 must be plugged into the INTELLEC DD MDS system with the power-protect switch enabled. Selection of a base address must follow the constraints as specified in Section C of this chapter. If these procedures are not followed, the program will not execute reliably (the program has no way of knowing where the MBB-80 controller has been physically placed in the address space or if it is correctly powered up).

The program will then begin the testing of every byte on the MBB-80 board. Each device will be tested, in turn, by writing and then reading back a random pattern (byte) one

page at a time. As each device is finished, a message so indicating will be printed. Once all devices on the board have been tested, a summary of errors (if any) for that pass will be listed and testing will automatically continue. When the user wishes to discontinue testing, the keying in of any character followed by a carriage return will terminate testing at the completion of the current pass. Any errors encountered will be listed, indicating the bubble device number (0-7 hex), the bubble page number (000-280 hex), the byte number within the page (0-11 hex), the pattern written and the pattern read back (in error). The occurrence of an error does not halt testing. Testing is continuous until the user halts execution by console input. When the program is halted, control automatically returns to the CP/M-86 operating system.

DIAG86M.A86 is a multi-page mode, 8086 assembly language, diagnostic program for the MBB-80. Its purpose is to provide a production version of a diagnostic program which runs under CP/M-86 and which can also be used to verify the correct operation of an MBB-80 Bubbl-Board. DIAG86M is functionally equivalent to DIAG86S.A86, except that DIAG80 runs in multi-page mode and thus, executes approximately four times faster than DIAG86S.

This diagnostic is invcked by executing the DIAG86M.CMD file on the CP/M-86 system disk. This program presents the same messages as DIAG86S and all instructions relevant to DIAG86S apply to DIAG86M.

There are, however, some special notes regarding the execution of DIAG86M. As explained in Section D of this chapter, there are <u>two</u> possible versions of this program, differentiated by a conditional assembly switch. One version uses interrupts generated over the MULTIBUS, while the other uses the polled mcde which interrogates the status of the MBB-80 controller. The "sign on" message will indicate which version is running. Since the polled mode of operation is used in the final CP/M-86 interface, this version is found on the system disk. The MULTIBUS vectored interrupt versicn requires that the hardware modifications to the MBB-80 board's interrupt pins and the iSBC 86/12A's interrupt pins be made (as described in Section C of this chapter) before program execution begins.

DIAG86M.CMD is the primary tool fcr performing diagnostic testing of MBB-80 Bubbl-Boards. It also provides a method of performing acceptance tests of newly purchased MBE-80 Bubbl-Boards. The user-specified base address for the controller allows the testing of any MBB-80 that is currently plugged into the INTELLEC DD MDS system.

VI. CP/M-86 INTERFACE IMPLEMENTATION

A. BUBBLE DEVICE STORAGE ORGANIZATION

The CP/M-86 interface design consists of two parts: (1) the implementation of the MBB-80 such that it will be functionally equivalent to a floppy disk generated for the CP/M-86 operating system; and, (2) the generation of a basic input/output system (BIOS) for the CP/M-86 operating system to include any combination of disks and MBB-80 Bubbl-Boards. This section will describe how the MBB-80 Bubbl-Board logical interface is made to appear as a "standard" disk to the CP/M-86 operating system.

CP/M-86, as does any CP/M system, uses two parameters when communicating with disk devices: tracks and sectors. The MBB-80 uses two different parameters: pages and devices. The translation of the 18 byte, physical, bubble page to that of a 128 byte CP/M sector was described in Section D of Chapter V. This organization configured the MBB-80 as consisting of eight devices (0-7), each with 80 "sectors" (1-80) of 128 bytes/sector. The remaining problem is that of mapping a CP/M track and sector to a corresponding MBB-80 device number and an MBB-80 "sector" number.

The BIOS in CP/M-86 has provisions for declaring the number of sectors per track on a given disk, as well as the total capacity of that disk (which implicitly implies the number of tracks). It was decided that each MBB-80 "track" would consist of 26 sectors, which is equivalent to the number of sectors per track of a CP/M-formatted single-density disk. This guaranteed compatible, if not optimal, use of the built-in CP/M blocking routines which are designed for tracks that have 26 sectors (or multiples thereof).

Addressing each of the eight devices on the MBB-80 Bubbl-Board requires additional software in that each individual device must be <u>separately</u> addressed when accessed. Therefore, any logical storage organization that caused the overlapping of logical storage units from one physical device to the next would have required additional software and, thus, incur a performance degradation. Consequently, it was decided that any given MBB-80 "track" would be entirely contained on one device. Since there are 26 CP/M-86 sectors per track on a single-density disk and 80 "sectors" on an MBB-80 device, there are 3 "tracks" per device with 2 "sectors" not used (wasted) on each device. Since there are 8 devices on an MBB-80 board, the total

capacity of the MBB-80 used would be 78K bytes on 24 "tracks" with a total of 14K bytes not used (wasted). This final storage organization is shown in Figure 6.1.

A method for mapping to this logical organization from a CP/M-86 sector call or track call was needed. The track mapping was the simplest. Mathematically, the device number is computed as follows:

```
DN = TN div 3
where DN  = MBB-80 device number (0-7)
      TN  = CP/M-86 track number requested
      div = integer division (disregard remainder)
```

For reasons of efficiency, this translation was implemented with tables rather than with arithmetic computations at the assembly language level.

The sector mapping, however, presents a more complex problem. As can be seen in Figure 6.1, bubble "sector" numbers range from 1-80 contiguously, across three "tracks", on each MBB-80 device. CP/M-86 uses a range of sector numbers between 1 and 26 on <u>each</u> track for a single-density disk. Given a requested CP/M-86 sector and track number, the corresponding MBB-80 "sector" number is computed. Mathematically, the "sector" number is computed as follows:

```
SN = (26 * (TN mod 3)) + SEC
where SN  = MBB-80 "sector" number (1-78)
      TN  = CP/M-86 track number requested
      mod = modulo division (remainder)
      SEC = CP/M-86 sector number requested
```

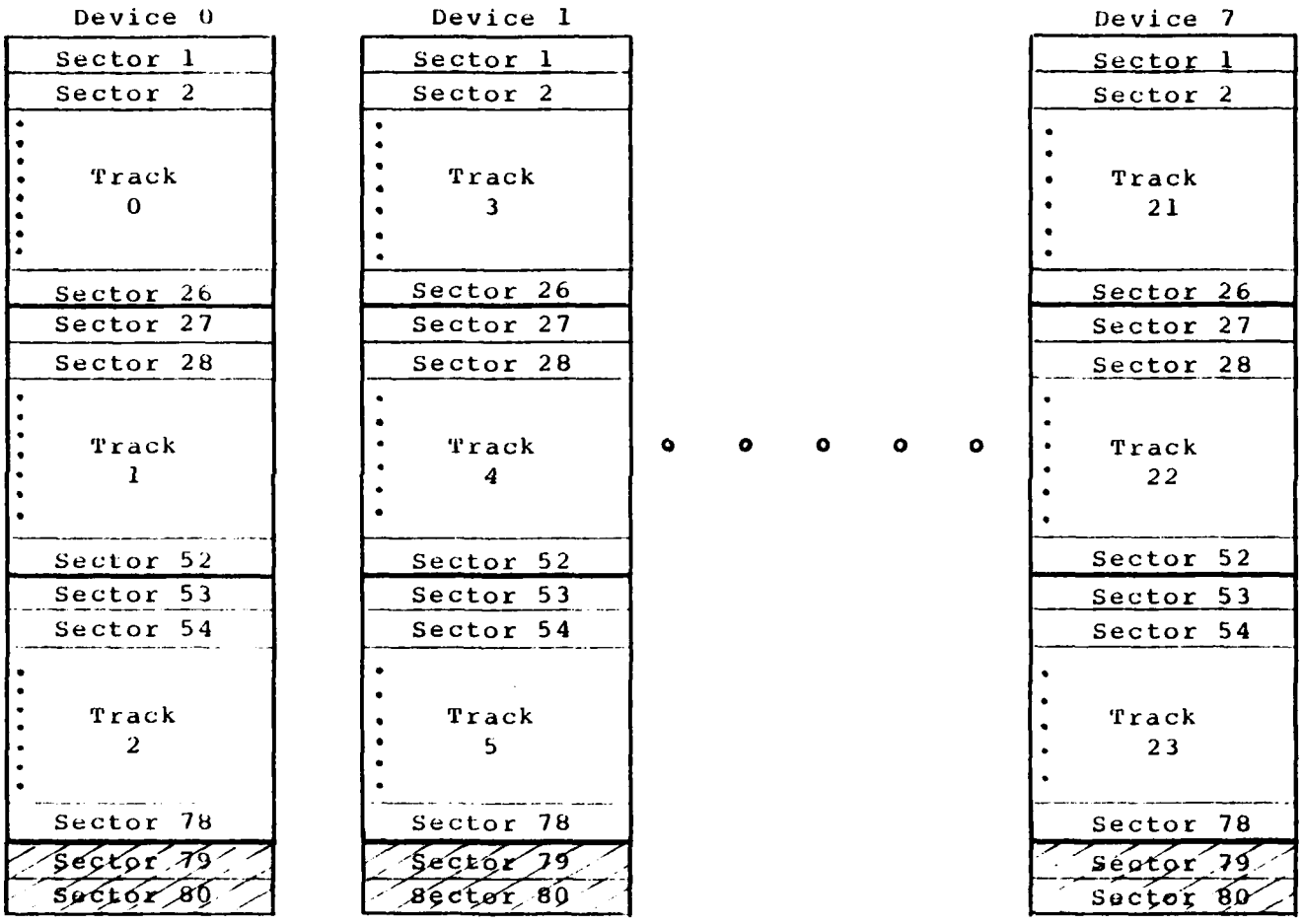


Figure 6.1 MBB-80 Logical Storage Organization

Again, for reasons of efficiency, this translation was implemented via tables rather than computed with the assembly language. The term "(26 * (TN mod 3))" is derived in the table lookup at the same time that the CP/M-86 track is being translated to a bubble device number.

Given an MBB-80 "sector" number (1-78), the physical, starting bubble page number can be computed (this routine was developed during and is explained in the Section D of Chapter V). For convenience, the formula for computing the physical, starting page number is repeated here:

```
SPN = ( (SN-1) * 12 ) mod 641
where SPN = starting page number (0-640)
      SN  = MBB-80 "sector" number (1-78)
      mod = modulo division (remainder)
```

The computation of the physical, starting page number was implemented with arithmetic statements and repetitive structures in the assembly language.

B. CP/M-86 BIOS CONSIDERATIONS

1. Structured Standards for the BIOS

The CP/M-86 operating system, as written by Digital Research, contains three parts: the Console Command Processor (CCP), the Basic Disk Operating System (BDOS) and the user-configurable Basic I/O System (BIOS). The CCP and BDOS portions of CP/M-86 occupy approximately 10K bytes and are distributed as a single hexadecimal code file (CPM.H86).

The CCP and BDOS communicate with physical devices via a well-defined interface in the BIOS. This interface is a set of call and return parameter conventions for the specific functions used when the CCP and BDOS communicate with the BICS. The BIOS contains all device-dependent code. A complete specification of the functional operation of the CCP and BDOS, along with the description of the BIOS interface, is contained in the CP/M-86 System Reference Guide (Reference 21). This section will describe the approach used in structuring a customized BIOS which provides an interface to both conventional CP/M-86 peripherals and the MBB-80 magnetic bubble device.

CP/M-86, as distributed by Digital Research, contains a sample, skeletal BIOS which can be utilized by a user to configure a customized BIOS. This skeletal BIOS is written in 8086 assembly language. A primary goal of this implementation is to provide a BIOS that can be easily modified and maintained. It was therefore considered essential to develop a BICS that consisted of structured, logically functional subroutines, within the constraints of the CP/M-86 physical component interface requirements. It was also considered necessary to provide adequate documentation within the program code. All subroutine input

and output parameters must be clearly defined. All modules that call a subroutine are listed in that called subroutine's documentation (in the code). The use of external branches out of a subroutine is not allowed and all subroutines terminate with a single "return" (viz., no subroutine is allowed to "fall through" to another section of code during execution). Naming conventions for constants, variables, labels and subroutines are consistent and meaningful and all identifiers are located in alphabetical order in logically-related sections for ease of location.

Although the above rules may result in some less-than-optimal execution structures from the viewpoint of speed, maintainability and ease of modification are essential goals. The primary purpose of this implementation of a BIOS, to provide a useable magnetic bubble system, can only be fully realized in a system that will allow for the custom modification of the implemented hardware and the supporting software.

2. Structured Approach to the BIOS

The CCP and BDOS portions of CP/M-86 are designed to interact with disks. Typically, an implementation of a specific disk unit, with a microcomputer running under

CP/M-86, involves only <u>one</u> kind of physical disk unit. This, of course, results in the simplest BIOS. However, the CCP and BDOS, in interacting with the BIOS via a standard interface, have a logical structure which will allow almost any combination of physical devices to be implemented in the BICS. The only requirement is that the BIOS preserve the standard interface to the rest of CP/M-86. It is this structural characteristic of the CP/M-86 operating system that was found to be very useful in this implementation.

The interface between the portions of CP/M-86 that are relevant to this implementation concern the "logical disk" interface. The CCP and BDOS are "aware" of up to 16 logical disks, which CP/M-86 will address via the parameters disk number, track and sector. It is this interface which must be preserved by any CP/M-86 BIOS implementation. Additionally, this BIOS must support the combination of standard floppy disk devices and MBB-80 Bubbl-Boards. Consequently, a structured approach is used within the BIOS itself for this implementation.

The BIOS is logically divided into four different areas: (1) standard CP/M-86 interface jump vectors; (2) subroutines which support communication with specific devices; (3) tables which define the physical characteristics and

configuration of the "disks"; and, (4) subroutines which operate (without modification) on those tables (even though the tables may be changed).

This approach provides a table-driven BIOS. A BIOS of this structure can be easily altered and allows for ease of configuration modification. Subroutines that provide specific device communications (viz., initialization, read a sector or write a sector) must be written for each type of device supported in the BIOS (a type is a specific double-density disk, hard disk, MBB-80, etc.). Tables are coded which describe the physical specifications of each logical CP/M-86 disk (viz., number of sectors, directories, capacity, etc.). Tables are also coded to provide the necessary information to support the mapping of logical CP/M-86 disk numbers to the required physical parameters for a particular type of device (viz., base addresses and internal disk numbers). These tables are fully described in Section D of this chapter.

Finally, the inclusion of all configuration-dependent information in the tables allows for ease of modification. Provided that no new device types are generated (which would require device-specific routines), the configuration (number and types of disks) can be changed entirely within the

tables without modifying the BIOS code itself. These tables are "included" into the BIOS code during assembly. A complete description of the BIOS generation will also be given in Section D of this chapter. All code in the BIOS which requires device-dependent information to perform its task will be designed to operate directly on the tables. This provides for a very modular implementation.

3. Jump Vector Interfaces

Entry to the BIOS from the CCP and BDOS is through a jump vector. The jump vector is a sequence of 21 three-byte jump instructions which transfer program control to the individual BIOS entry points (subroutines). Jump vector elements are in a standard order required by CP/M-86. Each BIOS entry point corresponds to a specific function, or task, to be performed by the BIOS for the CCP and BDOS. Each function has specific interface parameters (passed in designated registers) which must be adhered to in any BIOS implementation. All of these jump vectors, the BIOS entry points and their associated parameters are given on pages 56-64 of Reference 21.

Many of the functions in the BIOS need not be implemented and are simply coded as a "return" (i.e., the LISTOUT jump vector). Other functions deal with table "look

ups" within the BIOS on behalf of the CCP and BDOS. This section will be concerned with the jump vectors that require "knowledge" of specific physical disk devices. A complete description of the CP/M-86 jump vectors is found on pages 59-61 of Reference 21.

The "INIT" jump vector's function is to perform all initialization necessary for CP/M-86 that was not accomplished in the BOOT ROM or LOADER procedures. The "INIT" jump vector must be modified to perform all device initialization necessary. In this implementation, device initialization consists of calling a subroutine that performs initialization for all of the MBB-80 Bubbl-Boards that are logically and physically part of the system. Additionally, the default DMA address (20-bit, segment and offset) must be converted and stored as a 16-bit address for all devices that require a 16-bit address (viz., the iSBC 202 disk controller).

The jump vector called "SELDSK" has the function of selecting a disk for the next read or write. The BDOS call parameter is a logical disk number and the return parameter is the disk parameter header (DPH) for that device. The DPH is a standard table within CP/M-86 (BIOS) which describes the physical attributes of each disk and will be described

in Section D of this chapter. These basic functions were not altered. Additionally, however, upon selection of a CP/M-86 logical disk number, it is necessary to perform certain tasks. Given the logical disk number, a table is used to determine the type of device to which this disk number corresponds. If the device is a floppy disk, a mapping must be made to the physical disk number within the floppy disk controller (0-3 on the iSBC 202 double-density disk controller used in this implementation). If the device is an MBB-80, the base address for the memory-mapped I/O controller must be obtained. "SELDSK" must be modified to perform these functions by subroutine calls and to store this information for later use.

The jump vector called "HOME" has the function of moving a disk read head to its home position (track 0). There is no home position for the MBB-80 Bubbl-Board. Consequently, "HOME" must check the device type and if it is an MBB-80, the home request is translated into a request to set the track to zero (as required by CP/M-86).

The jump vector called "SETTRK" has the function of setting the track for the next read or write. The track number is passed in as a parameter. CP/M-86 supports track numbers in the range 0-65536. This allows the mapping of a

wide range of CP/M-86 track numbers directly to physical track numbers within disk controllers (viz., no translation). However, the MBB-80 storage organization requires the mapping of CP/M-86 track numbers to an MBB-80 device number and to a "sector" offset within that device. "SETTRK" must be modified to perform this function (by subroutine call) and to store this derived information for later use.

The "READ" and "WRITE" jump vectors have the function of performing a sector read (or write) to (from) the specified disk number at the specified track and sector. Normally, these vectors perform the actual operation directly by passing a channel command word to the disk controller for a single device. However, the MBB-80 requires entirely different routines to perform a read or write operation. Therefore, "READ" and "WRITE" must determine what type of device is currently being utilized and then call appropriate subroutines to perform MBB-80 reads and writes. The routines that actually perform the non-standard device (viz., MBB-80) read and write operations must also perform all necessary low-level mappings. In this implementation, the MBB-80 read and write subroutines will call on a sector translation subroutine that will map CP/M-86 sector numbers to MBB-80 "sector" numbers.

It should be noted that all device-specific details have been excluded from the jump vectors and coded within the device-specific subroutines. Jump vectors merely determine what type of device is being used (via tables) and then call appropriate subroutines. Although this BIOS implementation is specifically for the iSBC 202 disk controller and the MBE-80 Bubbl-Board (as the two types of logical disks), it can be easily modified to include any other type of disk device or magnetic bubble system as well. Operations that are dependent on a specific device type are isolated in specific subroutines. As described above, maintainability and ease of configuration modification have been designed into the structure of this BIOS implementation for CP/M-86.

C. USE OF THE CP/M-86 MBB-80 FORMAT PROGRAM

MB80FMT.A86 is a multi-page mode, 8086 assembly language program which formats the MBB-80 Bubbl-Board to meet IBM compatibility standards. This format is the required format for "new" CP/M-86 disks and consists of the hex pattern "E5" in every data byte of the disk. The program uses the multi-page polled mode to write the pattern to the MBB-80.

This format program is invoked by executing the MB80FMT.CMD file on the CP/M-86 system disk. The program will print appropriate messages and then request that the

user key in a four-digit, segment base address for the MBE-80 controller. Only four digits should be keyed in, followed by a carriage return. Keying in more or less than four digits, or invalid hex digits (viz., not in the range 0-F), will cause the printing of an error message and the user will then be asked to re-enter the segment base address. This segment base address consists of the high order 16 bits of the 20-bit address that is physically set on the MBB-80's address select pins. The address keyed in must match the MBB-80 controller's segment base address and the MBB-80 must be plugged into the INTELLEC DD MDS system with the power-protect switch enabled. Selection of a base address must follow the constraints as specified in Section C of Chapter V. If these procedures are not followed, the program will not execute reliably (the program has no way of knowing where the MBB-80 controller has been physically placed in the memory address space or if it is correctly powered up).

The program will then begin writing the hex pattern to every byte on the MBB-80 board. No further operator action is required. Each device (0-7) will be written to and, as each device is formatted, a message so indicating will be printed. Upon program completion, the "formatting complete"

message will be printed and control will return to the CP/M-86 operating system.

Since the polled mode is used to implement the multi-page mode of operation, there are no special considerations for running this program. The user-specified base address for the controller allows the formatting of any MBE-80 Bubbl-Board that is currently plugged into the INTELLEC DD MDS system. ME80FMT.CMD provides the only means of preparing an MBB-80 Bubbl-Board for use as a "disk" within the CP/M-86 operating system.

D. CP/M-86 BIOS IMPLEMENTATION

1. Modification of the Existing BIOS

The host CP/M-86 system, as described in Reference 18, contains a customized BIOS supporting a single iSBC 202 disk controller. This host BIOS is used to generate the LOADER BIOS as implemented in both the host system's BOOT ROM and LOADER program. The host BOOT ROM requires that a physical iSBC 202 disk be present in drive number 0 for boot loading (tracks 0 and 1). However, no restrictions exist as to the actual disk configuration that can be initialized and run by CPM.SYS (in its BIOS), which is read into RAM by the Loader program.

The basic routines for console input and output contained in the BIOS of Reference 18 were considered acceptable for use in this implementation. All other jump vectors either required modifications as described in the preceding section or were not considered to be consistent with the structured standards of this implementation. Consequently, all of the jump vectors were re-coded.

The device-dependent routines supporting the iSBC 202, found in Reference 18, were also incompatible with the structured standards and goals of this implementation. There was much redundancy and inefficiency in the algorithms and in the implementation as reflected in the code. In addition, the indexing method for mapping error codes to error messages for the iSBC 202 was found to be incorrect. Therefore, all routines relating to the iSBC 202 were re-written to perform correctly and to coincide with the standards and structured approach of this implementation. Obviously, the single iSBC 202 controller implementation of Reference 18 was limited to a single disk device. The implementation presented here is based on a table-driven BIOS that directly supports up to sixteen (the CP/M-86 maximum) disk drives which can be of two different types of devices. This necessitated the development of an entirely

new BIOS structure which resembles the BIOS of Reference 18 and the CP/M-86 distribution BIOS only in its preservation of the required jump vector interface standards.

2. **Disk Parameter Table**

The tables which determine the physical disk device characteristics of this CP/M-86 BIOS implementation are contained in two separate files. One file contains the specific device characteristics of each device, while the other file determines the currently generated configuration of disk devices.

The family of standard CP/M operating systems is designed to accept a table-driven specification for the physical characteristics of each logical CP/M disk device. These tables are called "disk definition tables" and consist of a disk parameter table for each disk generated as well as the scratchpad work areas for the operating system. The user is able to specify the number of logical disks to be generated (0-16), along with the characteristics of each disk (each having a separate entry). These characteristics include: the logical disk number, first and last sector number on each track, optional skew factor, blocksize, disk capacity, the number of directory entries, checked entries and the number of tracks to reserve for the operating

system. These parameters are specified in a file. Normally, the same type of device has the same parameters in every occurrence of that device type in the file. The only parameter that changes for devices of the same type is the logical disk number.

This file, containing the disk parameters, is used as input to a CP/M-86 utility program called GENDEF. This utility takes as input a file called filename.DEF and produces an 8086 assembly language source code file called filename.LIB. This output file contains the generated buffers, tables and scratch work areas needed by CP/M-86 to communicate with each disk device. A complete description of this disk parameter table generation and specification procedure is included on pages 65-73 of Reference 21.

The file generated by the GENDEF program is used in an ASM86 "include" statement (viz., inserted into the BIOS code) to be assembled within the BIOS. The disk parameter definitions (to be input to GENDEF) used for this implementation are included in the file DKPRM.DEF. This definition allows for three "disks": two iSBC 202 floppy disks and one MBB-80 "disk." If more or less disks are required, this disk parameter table must be changed and a new BIOS generated as described in a following section.

The disk definition parameters used in the BIOS of Reference 18 for the iSBC 202 controller were used in this implementation. The disk definition parameters used in this implementation for the MBB-80 were derived from the magnetic bubble storage organization scheme. First and last sector numbers were defined as 1 and 26, respectively. No skew translation was specified in that the BICS MBB-80 sector/track translation routines provide for this function. A blocksize of 1024 was defined so as to resemble a single-density disk. The capacity is 71K bytes as determined by the physical storage scheme and acccunting for reserved operating system tracks. Space was reserved for 32 directory entries, which allocates the minimum space possible for the MBB-80 directory. A checked entry of zero (0) is absolutely necessary to indicate that the MBB-80 is a non-removable media. Any directory checking will result in read-only status settings for the MBB-80 since CRC check-sum bytes are nct provided for by the MBB-80 ccntroller. Finally, two "tracks" are reserved for the operating system. This will aid in the implementation of an MBB-80 LOADER on track 0 and track 1.

3. Disk Configuration Tables

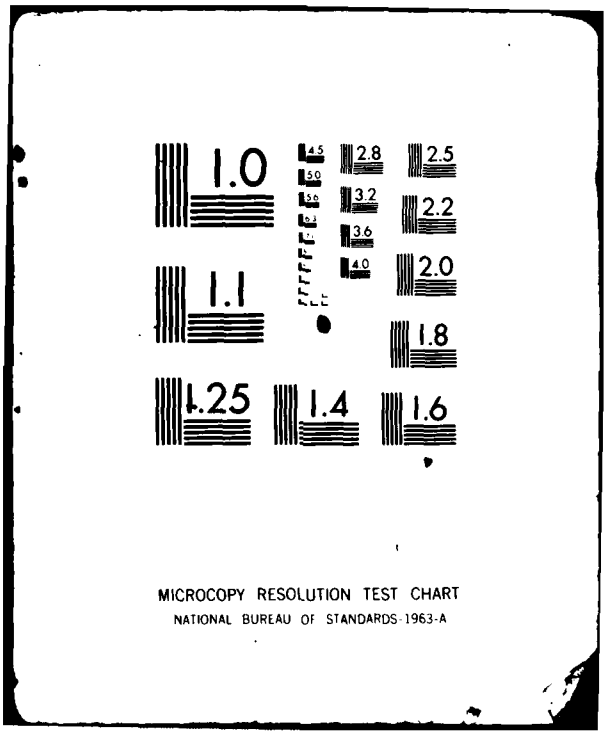
The DKPRM.DEF file contains information about the physical characteristics of each logical device. Since more than one possible device type may be generated in this implementation, it is necessary to map the CP/M-86 logical device numbers and their associated physical characteristics to the actual physical devices they represent. A set of tables has been developed to accomplish this task and is contained in the file called CONFIG.DEF. This file is also an 8086 assembly language source code file which is included into the BIOS during assembly. The configuration file is entirely a product of this implementation and has no relation to Digital Research's CP/M-86 distribution BIOS code. A summary description of the CONFIG.DEF file entries is contained in the CONFIG.DEF file itself. A complete discussion of the tables will be presented here.

The first entry in the configuration file is the number of logical disks defined. The identifier name in the file is "num_log_disk" and this entry is an equate statement. The value of this label can be in the range 0-16 decimal but must correspond to the "DISKS" statement in the DKPRM.DEF file.

AD-A115 028 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
ADAPTATION OF MAGNETIC BUBBLE MEMORY IN A STANDARD MICROCOMPUTE--ETC(U)
DEC 81 M S HICKLIN, J A NEUFELD

UNCLASSIFIED NL

2 3


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

The next entry is the device table. The identifier name in the file is "device_table" and this table is a 0-16 byte, one-byte per entry, table. This table describes the type of each disk device in logical order from CP/M-86 disk number zero (0) to the highest CP/M-86 disk number generated (which is "num_log_disks" minus 1). A byte position, or displacement, in the table corresponds to the logical CP/M-86 disk number (viz., byte offset 2 is the device type entry for CP/M-86 disk number 2, if generated). Each logical CP/M-86 disk that is defined must have an entry in this table indicating its device type. Therefore, the size of this table, in bytes, will equal the number of CP/M-86 disks defined. The different device types supported in this implementation each have a unique, hexadecimal, byte value to identify them. These codes are defined in equate statements at the beginning of the BIOS. The user will make entries into this table using the equate constants "disk_type" and "mbb80_type", with each successive entry separated by a comma.

Following the device table is the disk logical table for the iSBC 202 disk controller. The identifier name in the file is "DK_logical_table" and this table is a 0-16 byte, one-byte per entry, table. This table maps logical CP/M-86

disk numbers (0-15 possible) to internal iSBC 202 disk controller numbers. A single iSBC 202 controller can address up to four disks (internally numbered 0-3). A specific BIOS configuration may assign the four iSBC 202 disks to any four CP/M-86 disk numbers in the range 0-15. These CP/M-86 disk numbers must be mapped to iSBC 202 disk controller numbers (0-3) to be used in the disk channel command words. Therefore, this table maps logical CP/M-86 disk numbers to iSBC 202 disks (up to a maximum of four, since this implementation is designed for a <u>single</u> iSBC 202 controller). The size of this table, in bytes, can be up to 16 bytes, with the offset in the table corresponding to an entry for that CP/M-86 logical disk number. It is important to note that an entry must exist for all positions in the table up to and including the offset for the <u>last</u> CP/M-86 disk generated as an iSBC 202 disk device. The value "DK_null", which is merely a "place holder", is used for all entries which do <u>not</u> correspond to iSBC 202 disk devices.

For example, if two iSBC 202 disks were generated as logical CP/M-86 disk numbers 0 and 4, then the table would be five bytes long. Byte offsets 0 and 4 would contain 00H and 01H (as internal disk numbers) respectively, while byte offsets 1-3 would contain the "DK_null" place holding entry.

Byte offsets greater than 4, the last iSBC 202 disk generated in this example CP/M-86, need not be defined (ccded).

The last entry in the file is the MBB-80 logical table fcr the MBB-80 controller(s). The identifier name in the file is "MB_logical_table" and this table is a 0-16 word, one-wcrd per entry, table. This table maps logical CP/M-86 disk numbers (0-15 possible) to MBB-80 controller segment base addresses. Any number of MBB-80 "disks" may be generated anywhere (non-sequentially and non-contiguously) in the logical CP/M-86 disk range of 0-15. The size of this table, in words, must be exactly equal to the number of disks defined ("num_log_disks"). The word offset in the table corresponds to an entry (ccntroller segment base address) for that CP/M-86 MBB-80 "disk." It is important to note that an entry must exist for all positions in the table. The value "MB_null", which is merely a "place hclder", is used for all entries which do not correspond to an MBB-80 "disk" device. This table is also used to initialize the MBB-80 controller(s) based on the total number of CP/M-86 disks defined. The table is "walked through", with null entries being ignored and with non-null contrcller segment base addresses being initialized.

Therefore, unlike the disk logical table, there must be one entry for every logical CP/M-86 disk defined.

For example, if five CP/M-86 disks were generated, with numbers 0, 1 and 3 being iSBC 202 disks and numbers 2 and 4 being MBB-80 "disks", this table would be five words in length. Word offsets 2 and 4 would contain valid MBB-80 controller segment base addresses (in hex), while word offsets 0, 1 and 3 would contain the "MB_null" place holding entry. It is also important to note that when boot loading a CP/M-86 operating system with MBB-80 boards generated as disks, it is imperative that all MBB-80 boards be plugged into the INTELLEC MDS chassis and powered up. Failure to do so will cause the BIOS initialization routine to "hang" when processing the valid controller segment base addresses for MBB-80's in this table.

4. BIOS Generation Procedure

The procedure for the generation of a user-configured BIOS and a new CP/M-86 operating system is described on pages 80-82 of Reference 21. A synopsis of that procedure, along with the necessary modifications for this implementation, will be presented here.

The two files, DKPRM.DEF and CONFIG.DEF, are updated, as specified above, to reflect the user's desired devices and

configurations. The CP/M-86 GENDEF utility program is run utilizing DKPRM.DEF as input and producing DKPRM.LIB as output.

Assuming all necessary device-dependent modifications are made to the BIOS, assembly of the BIOS can take place. No modifications are necessary to this implementation BIOS if only iSBC 202 disks and MBP-80 "disks", in some combination, are to be used. This implementation's BIOS is included in the file called MBBIOS.A86 and is listed in Appendix E. In the code file MBBIOS.A86, there are the appropriate ASM86 "include statements" for the files DKPRM.LIB and CONFIG.DEF which will cause them to be inserted into MBBIOS.A86 during assembly. It was found that the 8086 cross assembler, a CP/M-80 program, has a small symbol table capacity. Therefore, assembly of MBBIOS.A86 must take place under CP/M-86.

Upon successful assembly, the file MBBIOS.H86 is produced. This file is concatenated to the CP/M-86 distribution CCP and BDOS, contained in the file CPM.H86, using the CP/M-86 utility program called PIP.CMD. The name of the resulting combined file should be a dummy, temporary name such as NEWCPM.H86. The resulting CCP, BDOS and customized BIOS hex file is then converted to the CMD file

format by executing the CP/M-86 utility program called GENCMD.CMD. The GENCMD options of an 8080 memory model and an absolute code location of "A40" must be specified. The format of the command with the options follows:

GENCMD NEWCPM 8080 code[A40]

Finally, the NEWCPM.CMD file is transferred to a new system disk that contains a LOADER program (see Chapter VII) and renamed to CPM.SYS. Now the tailoring process is complete and a boot load to the new system disk will invoke the CP/M-86 that has been generated.

5. Reconfiguring the BIOS

This implementation has been designed to directly support a single iSBC 202 disk controller and multiple MBE-80 boards in the BIOS. This allows for up to four (4) floppy disks and up to "n" (where "n" equals sixteen minus the number of iSBC 202 disks generated) MBB-80 disks.

The number and types of iSBC 202 and MBB-80 disks can be altered via the device and configuration tables. No changes are necessary to this implementation's BIOS code (MEBIOS.A86). Following the procedures of Section D.4 of this chapter will generate a new configuration in accordance with the information contained in the tables. Therefore, this BIOS can be easily expanded to support additional

MBB-80 "disks" and two more iSBC 202 drives (since the iSBC 202 controller is currently controlling only two physical drives).

This implementation has been generated with three (3) logical CP/M-86 disks. CP/M-86 disk numbers 0 (drive A:) and 2 (drive C:) map to the iSBC 202 controller's internal disk numbers 0 and 1. CP/M-86 disk number 1 (drive B:) maps to an MBB-80 Bubbl-Board controller at a segment base address of 08000H. A segment base address of 08000H was chosen for two reasons: (1) CP/M-86 I/O reserved addresses in the first 64K segment could not be used because of the inability to inhibit the onboard RAM for memory-mapped I/O, and (2) 080000H is significantly out of the address range for most applications. This address can be changed by modifying the entry in the CONFIG.DEF file for the MBB-80 controller segment base address.

E. EVALUATION OF THE IMPLEMENTATION

1. Performance

The primary criteria for the performance evaluation of this implementation was the speed of execution of the input/output functions of the types of disk devices. Three different programs were run on both an MBB-80 "disk" and on an iSBC 202 disk to determine execution times. A

conventional stopwatch was used for the timing and the results of those tests are summarized below.

The first test consisted of executing the CP/M-86 utility program, called PIP.CMD, which transfers CP/M-86 files between disks. The PIP program and target files of 2K, 6K and 28K bytes were loaded to both an MBB-80 "disk" and an iSBC 202 disk. Transfer operations were performed on each file on each device utilizing same-device resident copies of PIP, the target file and the destination file. The results of the test utilizing the PIP program were as follows:

File Size (Bytes)	MBB-80 (Seconds)	iSBC 202 (Seconds)
2K	3.5	11.2
6K	6.1	11.3
28K	18.2	21.2

The second test consisted of executing the CP/M-86 utility program, called ED.CMD, which is an object-oriented editor for files. The ED program and target files of 2K, 6K and 24K bytes were loaded to both an MBB-80 "disk" and an iSBC 202 disk. Edit operations were performed on each file on each device using same-device resident copies of ED, the target file and the destination file. The events timed and tested for an edit operation were the reading of the ED program into memory and the writing of the target file back

to its source disk from RAM memory. The results of the editing test were as follows:

File Size (Bytes)	MBB-80 (Seconds)		iSBC 202 (Seconds)	
	Read	Write	Read	Write
2K	2.6	1.5	8.4	5.3
6K	3.3	3.1	8.5	6.4
24K	3.4	10.4	8.7	13.9

The last test consisted of executing the CP/M-86 utility program, called ASM86.CMD, which assembles 8086 assembly language files into 8086 hex files. The ASM86 program and target files of 4K, 8K and 14K bytes were loaded to both an MBB-80 "disk" and an iSBC 202 disk. Assembly operations were performed on each file on each device utilizing same-device resident copies of ASM86, the target file and all of the ASM86 output files. The results of the assembly test were as follows:

File Size (Bytes)	MBB-80 (Seconds)	iSBC 202 (Seconds)
4K	20.9	28.4
8K	45.0	53.7
14K	64.3	81.9

From these test results it can be computed that an MBB-80 "disk" will provide an average increase of approximately 42 percent in input/output over an iSBC 202 disk. Of course, the more I/O intensive a program is, the greater the performance advantage that can be realized when using an MBB-80 vice an iSBC 202 disk.

2. Limitations

Three primary limitations were discovered in this implementation: transportability, density and transfer rate. A certain measure of transportability is provided in that any single MBB-80 Bubbl-Board is a logically complete CP/M-86 disk. The board can be removed from the INTELLEC DD MDS system chassis and moved to another system that supports MBB-80 devices under CP/M-86. However, this does require the "powering down" of the chassis prior to removing the board. It is also recognized that the media of a solid-state circuit board is different from that of a flexible, thin, magnetic disk. It is not clear which media is more conducive to transportability in any given application and environment.

The second limitation involves the relatively small capacity of the MBB-80 "disk" (78K bytes) in comparison to a single-density or double-density floppy disk (250K or 500K bytes). Even if the full capacity of the MBB-80 (92K bytes) could be used, the capacity difference is significant. The limited capacity of the MBB-80 restricts the number and size of the applications which can be executed entirely with the MBB-80 storage device. This limitation made large assemblies on MBB-80's and MBB-80 CP/M-86 resident disks impractical for a useful implementation.

The third limitation, transfer rate, becomes evident in viewing the test results presented in the performance section. As the size of the file is increased, the MBB-80's advantage over the iSBC 202 on I/O operations becomes less noticeable. This is primarily due to the fact that the MBB-80's transfer rate is only 45 Kbits/second, compared to a transfer rate of 250 Kbits/second for the iSBC 202. When I/O is performed where the number of seeks is relatively small in comparison to the number of actual bits transferred, the MBB-80's advantage is diminished. The validity of this trend could not be verified by the testing of large files because of the capacity limitation cited above.

It should be noted that, upon the availability of multiple MBB-80 boards, a system can be easily generated to support many MBB-80 "disks." Then, large applications could be run exclusively on MBB-80 "disks" by utilizing target disk specification parameters that are available in most CP/M-86 utility programs. Additionally, the future generation of a BIOS utilizing the currently available, high-capacity (1M byte) magnetic bubble devices is not to be precluded. This implementation of a BIOS provides an excellent and easily adapted framework for the addition of new types of disk devices.

3. Applications

This implementation of an MBB-80 Bubbl-Board within the CP/M-86 operating system has produced a workable host microcomputer environment which can be used for research and evaluation of magnetic bubble memory technology. It has also produced, with the subsequent addition of more MBB-80 boards, a developmental system which offers significant performance (speed of I/O) improvements over standard floppy disks in certain applications.

There is much theoretical research on the applicability of magnetic devices. The literature contains many untested and unimplemented designs, algorithms and programs for applications ranging from "fast sorts" to database management schemes. This implementation provides a host system capable of supporting research and experimentation in these areas on a fully-operational microcomputer system that supports magnetic bubble devices.

This implementation has produced a system capable of supporting up to sixteen MBB-80 "disks." Despite the individual capacity limit of 78K bytes per MBB-80, it is obvious that a significant reduction in program development time could be achieved utilizing exclusively MBB-80 logical "disks." This system is built upon the highly-regarded

Intel 8086, 16-bit microprocessor running under the CP/M-86 operating system. These characteristics, combined with the demcnstrated performance cf the MBB-80, contribute to prcvide a robust host system for research and application prcgram development utilizing magnetic bubble devices.

VII. BOOTLOADING CP/M-86 FROM THE MBB-80

A. BOOT ROM AND LOADER CONSIDERATIONS

When installed in the iSBC 86/12A, the BOOT ROM is part of the memory address space, beginning at byte location 0FE000H, and receives control when the system reset button is depressed. The BOOT ROM on the standard iSBC 86/12A contains the 957 monitor program as supplied by Intel. The program implemented on the EPROM chips was modified by adding code to the end of the 957 monitor program in memory addresses that were not utilized in the implementation of Reference 18. This customized addition of code to the 957 monitor program begins at memory address 0FFD40H and has the responsibility of reading the LOADER program from the first two system tracks of the CP/M-86 default disk drive into memory and then passing control to the LOADER program for execution.

The BOOT ROM is actually an EPROM which can be modified for specific implementations. The host development system, as described in Reference 18, reads the LOADER program from tracks 0 and 1 on physical drive number 0 of the iSBC 202 controller. The additional BOOT ROM code contains the necessary routines for initializing the iSBC 202 controller

and for reading the LOADER program from disk into memory. This procedure is initiated by issuing a "GFFD4:0" command to the 957 monitor, which passes control to the beginning of the bootstrap code in the BOOT ROM.

It was considered desirable to be able to boot load the CP/M-86 operating system from either an iSBC 202 disk or from an MBB-80 logical "disk." This requires two entry points into the additional code in the BOOT ROM. These entry points will set a flag indicating whether an iSBC 202 disk or the MBB-80 is to be used as the boot loading device. Additionally, routines for initializing the MBB-80 and for reading track 0 and track 1 on the MBB-80 had to be included in the BOOT ROM.

The available space in the BOOT ROM address space is severely limited. Therefore, the code for common functions in the BOOT ROM must be used by both an iSBC 202 boot request and an MBB-80 boot request when boot loading. Then, based on the value of the entry point flag, the requested device type (viz., iSBC 202 or MBB-80) initialization and read routines will be utilized to read into RAM the LOADER program from tracks 0 and 1 of the boot device. A common section of code will be used to pass control to the LOADER program for execution. A primary consideration must be

restricting the size of this additional code to the unused space after the 957 monitor program in the iSBC 86/12A's onboard EPROM.

The LOADER program is a simple subset of the CP/M-86 operating system that contains sufficient file processing capability to read CPM.SYS into memory from a system disk. When the LOADER program completes its operation, the CPM.SYS program receives control and proceeds to process operator input commands. The LOADER program consists of a loader CPM and a loader BDOS (distributed by Digital Research) along with a user-configured loader BIOS. The file resulting from the concatenation of these three modules is converted to an executable CMD file and placed on tracks 0 and 1 of the system disk. [Ref. 21: pp. 77-79]

A user-configured loader BIOS can be generated from the BIOS code developed in this implementation. The complete flexibility of device configuration that is possible in a standard BIOS is also possible in a loader BIOS. This implies an important consideration: the LOADER program does not have to read CPM.SYS from the same device that the LOADER program itself was read from. The LOADER program will read CPM.SYS from the default disk number and its corresponding device type based upon the device

configurations and mappings specified in the loader BIOS. Issuing a monitor "GO" command for the entry point of the iSBC 202 in the BOOT ROM will always result in the contents of tracks 0 and 1 (the LOADER program) on physical iSBC 202 drive number 0 being read into RAM. Likewise, issuing a monitor "GO" command for the entry point of the MBB-80 in the BOOT ROM will always result in the contents of "tracks" 0 and 1 of the MBB-80 at a controller segment base address of 08000H being read into RAM. The actual device configuration contained in the loader BIOS is not restricted by the type of device used by the BOOT ROM when reading the LOADER program.

B. BOOT ROM AND LOADER IMPLEMENTATION

The additional code for the BOOT ROM was written and tested. It provided for a conditional boot load from an iSBC 202 or from an MBB-80 at a controller segment base address of 08000H. The entry points are 0FFD40H for the iSBC 202 and 0FFD44H for the MBB-80. Upon depressing the reset button, the 957 monitor program begins execution. To boot load from the iSBC 202 the monitor command "GFFD4:0" is given, which is the same command as that used in the implementation of Reference 18. To boot load from the MBB-80, the monitor command "GFFD4:0004" is given.

The additional code for the BOOT ROM contains the entry points for the two device types, the iSBC 86/12A initialization procedures and the code necessary to initialize the selected boot device and read the LOADER program from the system tracks of that device. The additional code for the BOOT ROM is contained in the file called MB80ROM.A86. This file is assembled and the resulting object code is added to the 957 monitor program on the iSBC 86/12A's onboard EPROM. This procedure is described in Section C of this chapter.

The LOADER program itself consists of three parts: the Load CPM program (LDCPM.H86), the Loader Basic Disk Operating System (LDBDOS.H86) and the Loader Basic I/O System (LDBIOS.H86). The files LDCPM.H86 and LDBDOS.H86 are included as part of the standard Digital Research distribution system for CP/M-86. The loader BIOS is generated from the file MBBIOS.A86, which is also used to generate the standard CP/M-86 BIOS for this implementation. MBBIOS.A86 contains a conditional assembly switch, called "loader_bios", which, when enabled, produces a loader BIOS. The effect of this switch is to modify certain addresses to correspond to entry points into LDCPM and LDBDOS and to eliminate BIOS code that is not needed in the loader version of a BIOS.

The loader BIOS is configured in exactly the same manner as the BIOS itself and is fully described in Section D.4 of Chapter VI. The two files CONFIG.DEF and DKPRM.DEF must be modified to meet the user's requirements and to reflect the device that will contain CPM.SYS. It is the default drive, or CP/M-86 drive number 0, that is specified in the device table that determines which device will be searched for a CPM.SYS file.

The loader BIOS generation procedure is different from the BIOS generation procedure. Upon modification of the DEF files and successful assembly of MBBIOS.A86, a file called MBBIOS.H86 is produced. This file is concatenated to LDCPM.H86 and LDBDOS.H86 using the CP/M-86 utility program called PIP.CMD. The resulting combined file should be named LDBIOS.H86. The resulting loader CCP, BDOS and BIOS hex file is then converted to the CMD file format by executing the CP/M-86 utility program called GENCMD.CMD. The GENCMD options of an 8080 memory model and an absolute code location of "A400" must be specified. The format of this command is as follows:

```
GENCMD LDBIOS 8080 CODE[A400]
```

Finally, the new loader BIOS must be copied to tracks 0 and 1 of the new system disk. This is done by executing the

CP/M-86 utility program called LDCOPY.CMD. Assuming the loader BIOS executable file was called LDBIOS.CMD, the following comand would be used to initiate this process:

LDCOPY LDBIOS

The LDCOPY program will ask for a destination drive to receive the LDBIOS program on its track 0 and track 1. The target drive should have a scratch floppy disk (if an iSBC 202) or an MBB-80 board. A complete description of the LDCOPY procedure is given on pages 77-79 of Reference 21.

C. EPROM GENERATION

With the boot load program, MB80ROM, written, the only remaining task was the generation, or programming, of the required EPROM chips. The iSBC 86/12A has 8K bytes of onboard addressable EPROM, provided in four Intel 2716 EPROM chips of 2K bytes each. Because of the odd-even addressing of the iSBC 86/12A, two of the 2716s are devoted to the 4K even address bytes and the other two are devoted to the 4K odd address bytes. These even and odd address EPROMs are located at starting addresses 0FE000H and 0FE001H, respectively.

As previously mentioned, the 957 monitor program of the INTELLEC DD MDS system occupies a large portion of this onboard EPROM address space. The monitor occupies the

address space between 0FE000H and 0FFD22H and also has jump vectcrs located between 0FFFE0H and 0FFFFFH. The address space available for boot loader programs is approximately 720 (decimal) bytes between the end of the monitcr and the jump vectors. Since this available space is located entirely in the upper 4K bytes of the onboard EPROM, only the two 2716 EPROM chips containing the upper 4K bytes of address space need to be modified when incorporating a boot lcader.

Utilizing the CP/M-86 utility program called DDT.CMD, the contents of the upper 4K bytes of the iSBC 86/12A's onboard EPROM was read into memory and then saved as an executable CMD file. The INTELLEC DD MDS system was then reconfigured to the standard Intel 8080 sytem to facilitate the use of the ISIS cperating system and the Universal Prom Prcgrammer. The CP/M-80 utility program called DDT.COM was then utilized to replace the the existing bcot loader portion of the saved copy cf the EPROM contents with a copy of MB80ROM.CMD. This resulted in a single, complete, contiguous copy of the desired EPROM contents.

Intel 8080 assembly larguage programs were then written to split a file intc contiguous blocks of odd address and even address bytes. Using the CP/M-80 DDT program, the file

containing the new EPROM contents was loaded into memory and then each of the splitting programs loaded and executed. This resulted in the desired EPROM contents being divided into two contiguous blocks of 2K bytes each, one block containing the even address bytes of the split file and the other containing the odd address bytes of the file, and stored in RAM. The ISIS operating system was then booted with the two split blocks of the new EPROM contents still stored in RAM. The ISIS Universal PROM Mapper (UPM) system was then used to program two intel 2716 EPROM chips, one with the 2K byte contiguous block of odd address bytes and the second with the 2K bytes of even address bytes previously stored in RAM. The contents of the two newly programmed 2716 chips was then verified using the facilities of the UPM system.

The new EPROM chips, now containing MB80RCM.CMD in place of the boot loader provided by Reference 18, were then placed on the iSBC 86/12A and operationally tested. Boot loading from both an iSBC 202 disk and an MBB-80 "disk" was successfully accomplished. To ensure compatibility with the previous implementation of Reference 18, the CP/M-86 operating system of that implementation was successfully boot loaded with the new EPROM chips.

VIII. CONCLUSIONS

A. IMPLEMENTATION SYNOPSIS

All of the stated goals of this thesis were successfully accomplished in this implementation. A magnetic bubble device (MBB-80) was implemented utilizing a conventional microcomputer operating system (CP/M-86) and a commercial 16-bit microprocessor (Intel 8086). A fully operational system capable of testing, evaluating and utilizing a magnetic bubble device in a standard user environment was presented.

This implementation was accomplished in a manner such that future modifications and additions of hardware will be relatively easy. The hardware-dependent Basic I/O System (BIOS) of the CP/M-86 operating system was developed and coded as a structured, modularized, table-driven module. Device-dependent routines were isolated and confined to specific subroutines and tables. Device-independent code was structured to operate, without modification, utilizing the tables and subroutines which describe the specific hardware of the system. Documentation and structured programming techniques were emphasized to provide ease of program maintenance and modification.

This implementation provided a system in which the MBB-80 magnetic bubble device has the functional appearance of a disk to the CP/M-86 operating system. Consequently, at the user-interface level, no special considerations are necessary to utilize the magnetic bubble devices. Additionally, a system was generated consisting entirely of magnetic bubble devices. The system BOOT ROM and LOADER program were modified to show the feasibility of booting the CP/M-86 operating system from a magnetic bubble device. This produced a fully operational system supported only by magnetic bubble secondary storage (viz., no floppy disks).

This implementation and the proven feasibility of a system using magnetic bubble devices suggest many possible applications for this type of system. An operational system is now available for further testing and evaluation of magnetic bubble devices. The MBB-80, as a logical disk device generated into a CP/M-86 environment, becomes a compatible medium for different host systems (viz., hard disk, double-density, single-density). MBB-80 boards can be moved to any CP/M-86 MULTIBUS system, which has been generated with MBB-80 devices, and used to transfer files to the host system media.

B. RECOMMENDATIONS FOR FUTURE WORK

There are four major areas that present opportunities for future work. These areas are: (1) storage mapping schemes; (2) MBB-80 performance measurements; (3) generating and testing of new magnetic bubble devices; and, (4) implementation of new and existing applications utilizing MBE-80 devices.

The storage mapping scheme for the MBB-80, as implemented in this thesis, is both simple and efficient (viz., speed of code execution) but wastes 15.2 percent of the total capacity of the MBB-80 Bubbl-Board. Many storage schemes are possible if the MBE-80 is to be configured as a non-standard disk (viz., non-standard in relation to CP/M-86 track, sector and blocking schemes). It is not clear what physical configuration of the MBB-80, as logically presented to the CP/M-86 operating system, will provide the best tradeoff between speed and usable capacity for the MBB-80.

The performance evaluation of the MBB-80, as generated into CP/M-86 in this implementation, was limited to simple, timed tests of CP/M-86 utility operations. No attempts were made to perform an analytical evaluation of the low-level MBE-80 bubble operations in comparison to the corresponding low-level iSBC 202 disk operations. The MBB-80 low-level

diagnostic programs of Chapter V would provide an excellent vehicle for collecting data on the performance of low-level MBB-80 operations. Additionally, no evaluation was made of the operational and/or environmental ruggedness of the MBB-80. Much work is possible in determining the suitability of magnetic bubble devices for use in harsh environments. The fully operational magnetic bubble system will allow for testing and data collection under actual operating conditions.

The modularized, table-driven BIOS developed in this implementation is easily adapted to new hardware. Magnetic bubble devices based on new, high-density technology with parallel block/replicate architecture can be generated into the BIOS by simply adding appropriate device-dependent read/write routines and appropriate table entries. The framework provided by this implementation of a BIOS will lend itself to the addition of device types with a minimum amount of re-coding. The implementation of currently available 256K byte and 1M byte magnetic bubble devices into the CP/M-86 BIOS would provide a significant improvement in the usefulness of this implementation as a host development system.

Finally, this implementation of a BIOS can support multiple (up to 16) MBB-80 boards. With multiple boards (disks), this implementation system would be suitable for existing applications that utilize floppy disks. A total magnetic bubble system (without floppy disks) has been implemented with a single MBB-80 board. This allows the implementation of many applications on a total MBB-80 system where the availability or desirability of floppy disks is in doubt.

C. POTENTIAL APPLICATIONS

Chapter II and Chapter III presented evidence showing the current and future potential of magnetic bubble devices. The capacities, access rates and transfer rates of magnetic bubble devices are becoming competitive with, and often surpass, most conventional secondary storage media. Additionally, the characteristics of non-volatility, low power consumption, environmental ruggedness, high reliability and low maintenance exhibited by magnetic bubble devices give this technology a decided advantage over conventional secondary storage media in certain applications. Specifically, the application of magnetic bubble technology to the military environment appears very desirable.

Magnetic bubble devices require only DC power sources in the range of 1.0 amperes to 3.0 amperes at 5 volt and 12 volt levels. Power consumption is approximately 32 watts per megabyte of data capacity. Floppy disk devices require both AC and DC power sources. AC line frequency must be within one-half (1/2) hertz of the required frequency because of its effect on disk rotational speed and, thus, the read/write tolerances. DC power sources are in the range of 5.0 amperes to 8.0 amperes at 5 volt and 12 volt levels. Power consumption is approximately 350-400 watts per megabyte of data capacity. Magnetic bubble devices can operate in temperature ranges of 0 to 70 degrees Celsius and maintain data storage integrity in the range of -65 to 150 degrees Celsius. Magnetic devices can operate reliably in up to 100% relative humidity. Floppy disk devices can operate in temperature ranges of 10 to 40 degrees Celsius and at relative humidity levels between 20% and 80%. Operation of floppy disk devices outside these ranges can result in distortion of the diskette, followed by oxide deterioration, hygroscopic expansions, off-track recording and finally, irreversable magnetic effects. Magnetic bubble devices can withstand shock up to a 200G force and vibration up to a 20G force. No comparable figures for floppy and/or

hard disks are available since excessive shock and vibration are not considered as part of their potential "environments." Mean time between failure for magnetic devices is typically 5-10 years as compared to 5000-8000 hours (approximately 1 year) for floppy disk devices. It should be noted that disk devices, in general, require periodic maintenance and magnetic bubble devices do not.

Because of the stated advantages of magnetic bubble memory over other existing secondary storage technologies, it can be used in applications requiring mass storage of real time data that can be transferred to the system's main memory for processing. Most military applications have only the requirement for loading of programs and relatively small amounts of data to main memory. In these cases, the large capacity and transfer rate advantage of hard disks (relative to magnetic bubble devices) would not be needed. Consequently, magnetic bubble devices are a prime candidate for use in real time combat systems that must "go to war" such as the U.S. Navy's AEGIS weapons system.

Several specific military applications are currently using magnetic bubble devices. The Canadian Navy uses bubble memory for data recording at sea. The U.S. Air Force uses magnetic bubble cassettes to distribute and run F-15

aircraft maintenance diagnostic programs. Most military applications requiring a ruggedized storage medium are currently utilizing tape cassettes and flexible disk drives. Bubble memory, in portable cassette form, offers significant advantages over tape and disk media. A 2M bit bubble memory package, capable of operating in a temperature range of -54 to +155 degrees Celsius, is being developed for the Department of Defense by Western Electric and Bell Laboratories. It is targeted for use in a wide range of military applications. [Ref. 23: pp. 89-90]

It is apparent that there exists a significant need for magnetic bubble devices in military applications. Currently, the industry is addressing the problems of making magnetic bubble devices economically feasible, portable and more reliable. Even if the cost per bit remains higher than conventional media, the advantages of magnetic bubble devices in both military and commercial environments will present a convincing argument for the need and use of this technology.

APPENDIX A

PROGRAM LISTING OF DIAG80.ASM

```
;
;  FILENAMES: Pascal = MB.DIAG80.TEXT
;             CP/M = DIAG80.COM
;
;*************************************************************
;  8080 DIAGNOSTIC TEST FOR PC/M MBB-80 BUBBLE MEMORIES     *
;*************************************************************
;
;  CONFIGURATION:
;      HOST -  Intel 8080, 16 address lines, MDS system,
;              data bus on 8080 is eight bits.
;      MBB -   interrupts enabled, interrupts inhibited in
;              software, single-page mode, 20 address lines
;              decoding.
;
;  Simple bubble test for the 8080 - writes or reads one
;  user specified page at a time - user also specifies test
;  pattern if writing. Status register of MBB is displayed
;  to the console whenever used for debugging.
;
;  The MBB-80 controller base is defined by 'P$contbase'.
;  MBB-80 address select pins must correspond to this
;  address. This program uses memory mapped I/O through the
;  base address.
;
;*************************************************************
;  Jeffrey Neufeld and Michael Hicklin, CS-03, Thesis       *
;*************************************************************
;
;* Bdos function numbers for calls *
;
Bdos$conin      equ 01H         ;func # for Bdos read character
Bdos$conout     equ 02H         ;func # for Bdos write character
;
Bdos$entry      equ 0005H       ;entry for call to Bdos
Bdos$pstr       equ 09H         ;func # for Bdos print string
Bdos$reset      equ 00H         ;func # for CP/M-80 reset to CCP
;
;* Miscellaneous equates *
blank           equ 020H        ;Ascii blank
cr              equ 0dH         ;carriage return
eol             equ '$'         ;end of string char for pstr$func
lf              equ 0aH         ;line feed
;
;* MBB-80 characteristics (equates)
MB$maxpages     equ 641         ;# of pages on each bubble device
MB$pagesize     equ 18          ;bubble device page size
;
;* MBB-80 command byte masks
MB$busy$check equ 00100000B ;is cont busy? check (20H)
MB$init$cmd     equ 00000001B ;intialize the controller (01H)
MB$read$cmd     equ 10000010B ;single-page read cmd (82H)
MB$reset$cmd    equ 01000000B ;reset the controller (40H)
MB$write$cmd    equ 10000100B ;single-page write cmd (84H)
;
;* MBB-80 Controller and Ports
P$ccntbase      equ 04000H          ;base of controller
P$pselLo        equ P$contbase      ;page select lsb
P$pselHi        equ P$contbase+1    ;page select msb
```

```
P$cmdreg        equ P$contbase+2  ;command register
P$rdreg         equ P$contbase+3  ;read data register
P$wrreg         equ P$contbase+4  ;write data register
P$statreg       equ P$contbase+5  ;status register
P$lccpszlo      equ P$contbase+8  ;loop size lsb
P$lccpszhi      equ P$contbase+9  ;loop size msb
P$pgsize        equ P$ccntbase+12 ;page size register
P$selbub        equ P$contbase+15 ;bub dev select register
;
;*****************************************************
;                MAIN PROGRAM - DRIVER                *
;*****************************************************
;
                org  0100H
DIAG80:         lxi  SP,0b000H      ;stack pointer to app 44K
                di                  ;disable interrupts
                lxi  D,msg$signon   ;addr of signon msg
                call Print$String   ;print it
                call Init$Cont      ;init the MBB controller
                call Init$Devs      ;init the bubble devices
    Loop:       call Ask$User       ;user want read or write?
                cpi  'Q'            ;does user want to quit?
                jz   Quit           ;if sc, go quit
                push PSW            ;save user's answer
                call Get$Bubble     ;get user bubble # for test
                call Get$Page       ;get user page # for test
                pop  PSW            ;restore user's answer
                cpi  'R'            ;is this a read?
                jz   Read           ;if so, read   ; else=write
    Write:      call Get$Pattern    ;get user test pattern
                call Write$Page     ;write the page to MBB
                jmp  Loop           ;do until wants to quit
    Read:       call Read$Page      ;read back the page
                call Print$Out      ;write out results
                jmp  Loop           ;do until wants to quit
    Quit:       lxi  D,msg$quit     ;addr of done message
                call Print$String   ;print it
                mvi  C,Bdos$reset   ;func # to quit
                call Bdos$entry     ;call Bdos to terminate pgm
;
;**************** end of Main Program ********************
;
;
;
;*****************************************************
;*                ASK$USER  subroutine                *
;*****************************************************
                ;called from: Main.
Ask$User:       ;** asks user if wants read,write,cr quit
                ;** parm in - none.
                ;** parm out - ans in reg A,R=read,Q=quit
                ;                all else=write.
                lxi  D,msg$askfunc  ;addr of ask for func msg
                call Print$String   ;print it
                call Read$Char      ;get the user's answer
                push PSW            ;save user's answer
                call Crlf           ;skip a line after input
                pop  PSW            ;restore user's ans for ret
                ret
;
;
```

```
;*****************************************************
;*                  CRLF  subroutine                 *
;*****************************************************
                ;called from: Ask$User,Get$Bubble,Get$Page,
                ;             Get$Pattern, Print$Out.
Crlf:           ;** issues a carr ret, line feed to console
                ;** parm in - none.
                ;** parm out - none.
        mvi  A,cr               ;carr ret
        call Print$Char         ;output one char
        mvi  A,lf               ;line feed
        call Print$Char         ;output one char
        ret
;
;*****************************************************
;*                  GET$BUBBLE  subroutine           *
;*****************************************************
                ;called from: Main.
Get$Bubble:     ;** gets bubble # for test from console
                ;** parm in - none.
                ;** parm out - loads 'bubdev' variable.
        lxi  D,msg$getbub       ;addr of get-bubble msg
        call Print$String       ;print it
    ;get bubble number - one byte (0-7)
        call Get$Hex            ;get hex digit
        ani  0fh                ;clear high nibble
        lxi  D,bubdev           ;addr bubdev byte
        stax D                  ;store it
        call Crlf               ;skip a line after input
        ret
;
;*****************************************************
;*                  GET$HEX  subroutine              *
;*****************************************************
                ;called from: Get$Bubble, Get$Page,
                ;             Get$Pattern.
Get$Hex:        ;** gets a number from cons, converts both
                ;   nibbles to the hex value,ie., 'F' keyed
                ;   in = 46 Ascii, so FF returned in A
                ;** parm in - none.
                ;** parm out - double hex value in reg A.
        call Read$Char          ;get char from crt
        mvi  H,08H              ;high byte of table addr
        mov  L,A                ;low byte - index to table
        mov  A,M                ;table lookup
        ret
;
;*****************************************************
;*                  GET$PAGE  subroutine             *
;*****************************************************
                ;called from: Main.
Get$Page:       ;** gets user page # for test from console
                ;** parm in - none.
                ;** parm out - loads 'pageno' variable.
        lxi  D,msg$getpg        ;addr of getpage msg
        call Print$String       ;print it
    ;high byte of page number
        call Get$Hex            ;get hex digit
        ani  0fh                ;clear high nibble
        lxi  D,pageno$hi        ;addr pageno high
        stax D                  ;store it
    ;low byte - 2 ascii to 1 hex digit in pageno$lo
        call Get$Hex            ;get hex digit-hi
        ani  0f0H               ;clear low nibble
        mov  B,A                ;save high nibble
        push B                  ;save high
```

```
            call Get$Hex            ;get hex digit-lo
            ani  0fH                ;clear high nibble
            pop  B                  ;restore high
            ora  B                  ;combine hi and lo
            lxi  D,pageno$lc        ;addr pageno lcw
            stax D                  ;store it
            call Crlf               ;skip a line after input
            ret
;
;
;*****************************************************
;*                GET$PATTERN  subroutine            *
;*****************************************************
            ;called from: Main.
Get$Pattern:;** gets user pattern for test from console
            ;** parm in - none.
            ;** parm out - loads 'pattern' variable.
            lxi  D,msg$getpt    ;addr of get pattern msg
            call Print$String   ;print it
            call Get$Hex        ;get hex digit
            ani  0f0H           ;clear low nibble
            mov  B,A            ;save high nibble
            push B              ;save high
            call Get$Hex        ;get hex digit
            ani  0fH            ;clear high nibble
            pop  B              ;restore high nibble
            ora  B              ;combine hi and low
            lxi  D,pattern      ;addr of pattern
            stax D              ;store it
            call Crlf           ;skip lines after input
            call Crlf
            ret
;
;
;*****************************************************
;*                  INIT$CONT  subroutine            *
;*****************************************************
            ;called from: Main.
Init$Cont:  ;** inits the MBB controller
            ;** parm in - none.
            ;** parm out - none.
            lxi  D,msg$initc    ;addr of init msg
            call Print$String   ;print it
            lxi  B,MB$maxpages  ;pages in each loop
            lxi  H,P$loopsz1o   ;loopsize lsb port
            mov  M,C            ;load lsb of loopsize
            lxi  H,P$loopszhi   ;loopsize msb port
            mov  M,B            ;load msb of loopsize
            lxi  H,P$pgsize     ;page size port
            mvi  M,MB$pagesize  ;load page size
            lxi  H,P$cmdreg     ;command register port
            mvi  M,MB$reset$cmd ;issue reset command
            lxi  D,msg$dcnec    ;addr of done msg
            call Print$String   ;print it
            ret
;
;
;*****************************************************
;*                  INIT$DEVS  subroutine            *
;*****************************************************
            ;called from: Main.
Init$Devs:  ;** inits each bubble device on the MBB
            ;** parm in - none.
            ;** parm out - none.
            lxi  D,msg$initd    ;addr of init msg
            call Print$String   ;print it
            mvi  A,0            ;first device #
```

```
Each$dev:
        push PSW                ;save device #
        adi  030H               ;convert to ascii
        call Print$Char         ;print it
        lxi  D,msg$dev          ;addr of dev msg
        call Print$String       ;print it
        pop  PSW                ;restore dev #
        lxi  H,P$selbub         ;select bubble port
        mov  M,A                ;select this device
        push PSW                ;save dev #
        lxi  H,P$cmdreg         ;command register port
        mvi  M,MB$init$cmd      ;issue init command
        call Wait               ;let controller work
        lxi  D,msg$done         ;addr cont done msg
        call Print$String       ;print it
        pop  PSW                ;restore dev #
        inr  A                  ;next device #
        cpi  08H                ;last device ?
        jnz  Each$dev           ;if not, do next
        lxi  D,msg$doned        ;addr done msg
        call Print$String       ;print it
        ret
;
;*****************************************************
;*              LOAD$PAGE  subroutine                *
;*****************************************************
                ;called from: Read$Page, Write$Page.
Load$Page:      ;** loads the variable 'pageno' to the MBB
                ;** parm in - none.
                ;** parm out - none.
        lxi  H,pageno$lo        ;addr of page # lsb
        mov  A,M                ;to accum
        lxi  D,P$psello         ;page select lsb port
        stax D                  ;load it
        inx  H                  ;to page # msb
        inx  D                  ;to page select msb port
        mov  A,M                ;to accum
        stax D                  ;load it
        ret
;
;*****************************************************
;*              PRINT$CHAR  subroutine               *
;*****************************************************
                ;called from: Crlf,Init$Devs,Print$1,
                ;             Print$2.
Print$Char:     ;** calls Bdos to write a char to console
                ;** parm in - char to write in Reg A.
                ;** parm out - none.
        mov  E,A                ;load parm for Bdos
        mvi  C,Bdos$concut      ;func # for Bdos write char
        push PSW
        call Bdos$entry         ;call Bdos to write
        pop  PSW
        ret
;
;*****************************************************
;*              PRINT$OUT  subroutine                *
;*****************************************************
                ;called from: Main.
Print$Out:      ;** reads page from MBB buf-writes to cons
                ;** parm in - none.
                ;** parm out - none.
        lxi  D,msg$prt          ;addr of print out msg
        call Print$String       ;print it
        mvi  C,MB$pagesize      ;counter for bytes to read
```

```
Prt:    lxi  D,P$rdreg      ;read data register port
        ldax D              ;load from fifo to accum
        push B              ;save counter
        call Print$2        ;print what was read
        pop  B              ;restore counter
        dcr  C              ;dec counter
        jnz  Prt            ;read next if not 18D read
        lxi  D,msg$done     ;addr of done msg
        call Print$String   ;print it
        call Crlf           ;skip a line
        ret
;
;*****************************************************
;*                PRINT$STRING subroutine            *
;*****************************************************
                ;called from: Ask$User,Get$Bubble,Get$Page,
                ;   Get$Pattern,Init$Cont,Init$Devs,Main,
                ;   Print$Cut,Read$Page,Write$Page.
Print$String:   ;** prints a string to console via Bdos.
                ;** parm in - address of string in reg D.
                ;** parm out - none.
        mvi  C,Bdos$pstr    ;func# for Bdos print string
        push PSW
        call Bdos$entry     ;call Bdos to print
        pop  PSW
        ret
;
;*****************************************************
;*                  PRINT$1 subroutine               *
;*****************************************************
                ;called from: Print$2.
Print$1:        ;** converts hex value of low nibble to
                ;   Ascii and prints it to console.
                ;** parm in - hex value to print in reg A.
                ;** parm out - none.
        ani  0fH            ;clear high nibble
        adi  090H           ;convert hi
        daa
        aci  040H           ;convert lo
        daa
        call Print$Char     ;print char
        ret
;
;*****************************************************
;*                  PRINT$2 subroutine               *
;*****************************************************
                ;called from: Print$Out,Wait.
Print$2:        ;** converts one byte hex to two Ascii
                ;   digits and prints out one at a time.
                ;** parm in - hex value to print in reg A.
                ;** parm out - none.
        push PSW            ;save low digit
        rrc!rrc!rrc!rrc!    ;move hi nibble to low
        call Print$1        ;convert and print
        pop  PSW            ;restore low digit
        call Print$1        ;convert and print
        mvi  A,blank        ;blank char
        call Print$Char     ;print it for separation
        ret
;
;
```

```
;*****************************************************
;*                READ$CHAR  subroutine              *
;*****************************************************
                ;called from: Ask$User, Get$Hex.
Read$Char:      ;** reads one character from the console
                ;** parm in - none.
                ;** parm out - char read in reg A.
          mvi   C,Bdos$conin    ;func # for Bdos read char
          call  Bdos$entry      ;call Bdos to read
          ani   07fH            ;clear parity bit
          ret
;
;*****************************************************
;*                READ$PAGE  subroutine              *
;*****************************************************
                ;called from: Main.
Read$Page:      ;** interfaces with MBB to read a page
                ;** parm in - uses 'pageno' & 'bubdev' vars
                ;** parm out - none.
          call  Load$Page       ;load page number to MBB
        ;load bubble device number
          lxi   D,bubdev        ;load addr of dev #
          ldax  D               ;to accum
          lxi   H,P$selbub      ;select bubble register port
          mov   M,A             ;load dev #
        ;issue read command
          lxi   D,msg$rd        ;addr of reading msg
          call  Print$String    ;print it
          lxi   H,P$cmdreg      ;command register port
          mvi   M,MB$read$cmd   ;issue read command
          call  Wait            ;let controller work
          lxi   D,msg$done      ;addr of done msg
          call  Print$String    ;print it
          ret
;
;*****************************************************
;*                WAIT  subroutine                   *
;*****************************************************
                ;called from: Init$Devs,Read$Page,
                ;             Write$Page.
Wait:           ;** makes a delay while the cont works
                ;** parm in - none.
                ;** parm out - none.
          lhld  0               ;30 cycle delay at 2.5MHz
          lhld  0               ;5 cycles each lhld inst
          lhld  0
          lhld  0
          lhld  0
   Wait1: lxi   H,P$statreg     ;status register port
          mov   A,M             ;read status register
          push  PSW             ;save status
          call  Print$2         ;print out status
          pop   PSW             ;restore status
          ani   MB$busy$check   ;busy mask check
          jnz   Wait1           ;if busy, check again
          lxi   H,P$statreg     ;stat reg port-get last stat
          mov   A,M             ;read status register
          call  Print$2         ;print out status
          ret
;
;
```

```
;*****************************************************
;*                WRITE$PAGE  subroutine             *
;*****************************************************
                ;called from: Main.
Write$Page:     ;** interfaces with the MBB tc write a page
                ;** parm in - uses 'pageno' & 'bubdev' vars
                ;** parm out - none.
          call  Load$Page       ;load page number tc MBB
       ;load 18 test bytes to fifo
          mvi   C,MB$pagesize   ;counter for bytes (18D)
          lxi   D,P$wrreg       ;write data register port
          lxi   H,pattern       ;addr cf pattern tc write
          mov   A,M             ;load pattern to accum
  Write1: stax  D               ;write a byte to fifo
          dcr   C               ;dec counter
          jnz   Write1          ;jump if not 18D written
       ;load bubble device number
          lxi   D,bubdev        ;lcad addr of dev #
          ldax  D               ;to accum
          lxi   H,P$selbub      ;select bubble register port
          mov   M,A             ;load dev #
       ;issue write command
          lxi   D,msg$wrt       ;addr cf writing msg
          call  Print$String    ;print it
          lxi   H,P$cmdreg      ;command register pcrt
          mvi   M,MB$write$cmd  ;issue write command
          call  Wait            ;let controller work
          lxi   D,msg$done      ;addr cf done msg
          call  Print$String    ;print it
          ret
;
;*****************************************************
;                 DATA AND VARIABLE AREA              *
;*****************************************************
;
bubdev      db  0
pagenc$lo   db  0
pageno$hi   db  0
pattern     db  0
;
msg$askfunc db  'Enter a R to read, Q to quit, all else '
            db  'is write: ',ecl
msg$dev     db  ' device # initing. ',eol
msg$done    db  ' done.',cr,lf,eol
msg$donec   db  'Done with controller.',cr,lf,eol
msg$doned   db  'Done with devices.',cr,lf,cr,lf,ecl
msg$getbub  db  'Input 1 digit bubble # (0-7): ',ecl
msg$getpg   db  'Input 3 digit hex page # (000-280): ',eol
msg$getpt   db  'Input 2 digit hex test pattern (00-FF): '
            db  eol
msg$initc   db  'Initializing ccntroller.... ',eol
msg$initd   db  'Initializing the devices... ',cr,lf,eol
msg$prt     db  'Page read is: ',eol
msg$quit    db  lf,cr,lf,cr,'** End of Test **',lf,cr,eol
msg$rd      db  'Reading a page... ',eol
msg$signon  db  lf,cr,'                ** MBB-80 CP/M-80 '
            db  'DIAGNOSTIC TEST **',cr,lf,cr,lf,ecl
msg$wrt     db  'Writing a page... ',eol
;
```

```
;table for converting ascii to hexadecimal
org  0830H
             db 00H,11H,22H,33H,44H,55H,66H,77H,88H,99H
org  0841H
             db 0aaH,0bbH,0ccH,0ddH,0eeH,0ffH
;
;*************************************************************
;                  End of Prcgram                            *
;*************************************************************
             END     0100H
;
```

APPENDIX B

PROGRAM LISTING OF DIAG86S.A86

```
;
;  FILENAMES: Pascal = MB.DIAG86S.TEXT
;             CP/M = DIAG86S.CMD
;
;*************************************************************
;*   8086 DIAGNOSTIC TEST FOR PC/M MBB-80 BUBBLE MEMORIES    *
;*************************************************************
;
;  CONFIGURATION:
;      HOST - Intel 86/12A SBC, 20 address lines, MDS system,
;             Data bus on 86/12A converting to low 8 bits
;             all high.
;      MBB -  interrupts inhibited, single-page mode,
;             20 address lines.
;
;  This program writes and then reads a test pattern in
;  each page of each bubble chip on MBB-80 boards. Error
;  diagnostics are printed as errors are found. An error
;  log is printed at the end of each pass. Testing is
;  continuous until any character is keyed into the console.
;
;  The MBB-80 controller base address is read into variable
;  'MB_contbase'. MBB-80 address select pins must correspond
;  to this address. This program uses memory mapped I/O
;  through the base address.
;
;*************************************************************
;*   Jeffrey Neufeld and Michael Hicklin,  CS-03,  Thesis    *
;*************************************************************
;
;* Bdos function numbers for calls *
Bdos_conbuf    equ 10      ;console input string funct #
Bdos_conout    equ 2       ;console output char funct #
Bdos_constat   equ 11      ;get console status funct #
Bdos_pstring   equ 9       ;print string until '$' funct #
Bdos_reset     equ 0       ;CP/M-86 reset to CCP funct #
;
;* MBB characteristics *
MB_buflen      equ 18      ;buffer length for single page
MB_maxdevs     equ 7       ;bubble devices are #0-#7
MB_maxpages    equ 641     ;# of pages on each bubble device
MB_pagesize    equ 18      ;bubble device page size
;
;* MBB command byte masks (with interrupts inhibited) *
MB_busy_check  equ 00100000B ;cont busy? status check (20H)
MB_init_cmd    equ 10000001B ;initialize the controller (81H)
MB_read_cmd    equ 10000010B ;single-page read command (82H)
MB_reset_cmd   equ 11000000B ;reset the controller (C0H)
MB_write_cmd   equ 10000100B ;single-page write command (84H)
;
;* Miscellaneous equates *
blank          equ 020H    ;Ascii blank
conbuf_size    equ 80      ;size for input buffer for console
cr             equ 0dH     ;Ascii carriage return control char
lf             equ 0aH     ;Ascii line feed control char
;
```

```
;*************************************************************
;*                  MAIN PROGRAM - DRIVER                    *
;*************************************************************
;
        CSEG
;
DIAG86S:  call Set_Up             ;do initialization
          call Get_Cont_Addr      ;get address of MBB-80 base
          call Init_Cont          ;init the cont and devices
  Test_loop:
          call Get_Test_Buffer ;get a test pattern,fill buff
          call Write_Page         ;write a page to bubble
          call Read_Page          ;read a page from bubble
          call Check_Errors       ;check errors in write/read
       ;advance to next page in a device, see if last page
          inc  curr_page_no       ;increment current page #
          cmp  curr_page_no,MB_maxpages-1 ;last page on dev?
          jnz  Test_loop          ;if not, test next page
       ;was last page,advance to next bubble device on board
          mov  DX,offset msg_donebub ;addr of done bub msg
          call Print_String       ;write msg to conscle
          cmp  curr_bub_no,MB_maxdevs ;last bubble on board?
          jz   Done_pass          ;if sc, done with a pass
       ;prepare to test next bubble device
          inc  curr_bub_no        ;if nct, increment device #
          mov  curr_page_no,0     ;set page number back tc zero
          inc  errptr             ;ptr tc next entry (dev)
          jmp  Test_loop          ;go test next device
       ;finished with all devices on board, print summary
       ;prepare to run another pass if not stopped by user
  Dcne_pass:
          call Error_Summary      ;print error summary
          call End_Pass           ;end of pass housekeeping
       ;see if anything keyed in at the console
          mov  CL,Bdcs_constat ;function # for Bdcs call
          call Bdos               ;call Bdos to get ccns status
          cmp  AL,01              ;01=char keyed in, 00=nothing
          jz   Done_test          ;something keyed, user quits
       ;user wants to continue
          mov  DX,offset msg_testing ;addr of testing msg
          call Print_String       ;write msg to console
          jmp  Test_loop          ;keep testing
       ;user wanted to quit the testing
  Dcne_test:
          call Close_Up           ;do end of run housekeeping
          mov  CL,Bdcs_reset      ;function # for Bdcs call
          mov  DL,0               ;parameter to release memory
          call Bdos               ;call Bdos to terminate prog
;
;************** end of Main Prcgram **********************
;
;
;
;*************************************************************
;*                  BDOS (CPM/86)  subroutine                *
;*************************************************************
                ;called from: Close_Up,Main,Get_Cont_Addr,
                ;             Print_String, Putchar.
Bdcs:           ;** entry tc Bdcs via software interrupt 224
                ;** parm in - caller loads regs as per req
                ;** parm out - as supplied by Bdos returns
          int  224              ;8086 software interrupt
          ret
;
;
```

```
;*****************************************************
;*                CHECK_ERRORS  subroutine           *
;*****************************************************
                ;called from: Main.
Check_Errors:   ;** see if read what was written
                ;** parm in - none
                ;** parm out - none
          mov   AL,pattern      ;pattern to accum for manipul
          mov   CX,MB buflen    ;counter for loop thru buffer
          mov   BX,offset test_buffer ;index into test buffer
  Test_byte:
          cmp   [BX],AL         ;compare buff to pattern
          jz    Good_test       ;if good, check next byte
          push  AX!push BX!push CX ;save patt/buff addr/cntr
          call  Err_Out         ;it is bad, print error
          call  Log_Error       ;           log error
          pop   CX!pop BX!pop AX ;restore cntr/buff addr/patt
  Good_test:
          inc   BX              ;increment index
          loop  Test_byte       ;dec CX and loop if not zero
          ret
;
;*****************************************************
;*                CLOSE_UP  subroutine               *
;*****************************************************
                ;called from: Main.
Close_Up:       ;** reads garbage from console,issues goodbye
                ;** parm in - none
                ;** parm out - none
       ;clear stop input characters from the console buffer
          mov   CL,Bdos_conbuf  ;input console string func#
          mov   BX,offset cons_buff ;area for cons input
          mov   byte ptr [BX],conbuf_size;tell Bdos buff size
          mov   DX,BX           ;load parameter reg for Bdos
          call  Bdos            ;read the console
       ;issue the goodbye message
          call  Crlf            ;skip extra line
          mov   DX,offset msg_endtest ;addr of end test msg
          call  Print_String    ;write msg to console
          ret
;
;*****************************************************
;*                CRLF  subroutine                   *
;*****************************************************
                ;called from: Close_Up,Get_Cont_Addr,
                ;End_Pass,Init_Cont,Main,Print_String,Set_Up.
Crlf:           ;** sends carriage return,line feed to cons
                ;** parm in - none
                ;** parm out - none
          mov   AL,cr           ;carriage return char
          call  Putchar         ;write it to console
          mov   AL,lf           ;line feed char
          call  Putchar         ;write it to console
          ret
;
;*****************************************************
;*                END_PASS  subroutine               *
;*****************************************************
                ;called from: Main.
End_Pass:       ;** performs end of pass housekeeping
                ;** parm in - none
                ;** parm out - none, effects global vars
       ;convert pass # to Ascii and print after pass message
          mov   AL,pass_no      ;pass number to accum
          call  Hex_To_Ascii    ;convert to Ascii
```

```
            mov  BX,offset msg_d_pass;addr of pass # in msg
            mov  byte ptr [BX],DH ;load high byte tc msg
            inc  BX                ;bump to next position in msg
            mov  byte ptr [BX],DL ;load low byte to msg
            mov  DX,offset msg_dcnepass ;addr of done pass msg
            call Print_String      ;write msg to conscle
            call Crlf              ;skip a line
        ;inc pass number and reset all variables fcr new pass
            inc  pass_no           ;add one to pass number
            mov  newpass_flag,1    ;set new-pass flag cn
            mov  curr_bub_no,0     ;reset to bubble device 0
            mov  curr_page_no,0    ;reset page number to 0
            mov  errptr,offset errlcg ;reset addr of error log
            ret
;
;*****************************************************
;*                ERR_OUT  subroutine                *
;*****************************************************
                ;called from: Check_Errors.
Err_Out:        ;** issue an error message to the ccnsole
                ;** parm in - BX addr in buff cf byte error
                ;** parm out - none. effects global vars
            push BX ! push BX      ;save addr of error twice
            cmp  newpass_flag,1    ;is this a new pass ?
            jnz  Prt_err           ;if nct, print error now
            mov  newpass_flag,0    ;turn flag off
            mov  DX,offset msg_header ;load addr of header
            call Print_String      ;print the header
        ;put zeros into all error ccunts in the log
            mov  CX,MB_maxdevs+1 ;count for # of dev to loop
            mov  BX,offset errlog ;addr of error log
  Clr_log:
            mov  byte ptr [BX],0 ;clear log entry error count
            inc  BX                ;bump pointer to next entry
            loop Clr_log           ;dec CX and loop if not zero
  Prt_err:
            mov  AL,curr_bub_no    ;bub dev # to accum
            call Hex_To_Ascii      ;ccnvert to Ascii
            mov  msg_e_dev,DH      ;move in high byte to msg
            mov  msg_e_dev+1,DL    ;move in low byte to msg
        ;load page number of error
            mov  AL,byte ptr curr_page_no+1;hi byte cf  page#
            call Hex_To_Ascii      ;convert to Ascii
            mov  msg_e_page,DH     ;high byte to msg(dig 1)
            mov  msg_e_page+1,DL ;low byte to msg(dig 1)
            mov  AL,byte ptr curr_page_no ;lo byte cf page#
            call Hex_To_Ascii      ;ccnvert to Ascii
            mov  msg_e_page+2,DH ;high byte to msg(dig 2)
            mov  msg_e_page+3,DL ;low byte to msg(dig 2)
        ;compute and load byte cffset of error in page
            pop  BX                ;restore addr err byte offset
            addr_buff equ cffset test_buffer ;fcr ccmputation
            sub  BX,addr_buff      ;compute err offset in buff
            mov  AL,BL             ;offset to AL for ccnversion
            call Hex_To_Ascii      ;convert to Ascii
            mov  msg_e_byte,DH     ;move in nign byte to msg
            mov  msg_e_byte+1,DL ;move in low byte to msg
        ;load pattern that was written and what was read back
            mov  AL,pattern        ;load pattern just written
            call Hex_To_Ascii      ;ccnvert to Ascii
            mov  msg_e_wrote,DH    ;move in high byte to msg
            mov  msg_e_wrote+1,DL ;move in low byte to msg
            pop  BX                ;restore addr cf err offset
            mov  AL,[BX]           ;load byte just read back
            call Hex_To_Ascii      ;convert to Ascii
            mov  msg_e_read,DH     ;mcve in high byte to msg
            mov  msg_e_read+1,DL ;move in low byte to msg
```

```
            mov  DX, offset msg_err ;addr of total error msg
            call Print_String       ;print the error message
            ret
;
;*************************************************************
;*                  ERROR SUMMARY  subroutine                *
;*************************************************************
                 ;called from: Main.
Error_Summary:   ;** outputs summary of errors on each device
                 ;** parm in - none
                 ;** parm out - none
            mov  DX,offset msg_summary ;addr of summary msg
            call Print_String       ;write msg to conscle
        ;step thru errlog-convert to Ascii - print err counts
            mov  CX,MB_maxdevs+1 ;count for loop - # of devs
            mov  BX,offset errlog ;addr of error log
            mov  DI,offset msg_counts ;addr of msg sum counts
    prt_loop:
            mov  AL,[BX]            ;get count from error log
            push BX!push CX!push DI ;save addr,counter,index
            call Hex_To_Ascii       ;convert to Ascii
            pop  DI! pop CX! pop BX ;rest index,counter,addr
            mov  byte ptr [DI],DH ;load high byte to msg
            inc  DI                 ;bump to next pos in msg
            mov  byte ptr [DI],DL ;load low byte to msg
            inc  DI                 ;bump to next pos in msg
            mov  byte ptr [DI],blank ;Ascii blank to msg
            inc  DI                 ;bump to next pos in msg
            inc  BX                 ;increment buff addr to next
            loop prt_loop           ;dec CX and loop if not zero
            mov  DX,offset msg_counts ;addr of msg sum counts
            call Print_String       ;write msg to conscle
            ret
;
;*************************************************************
;*                  GET CONT ADDR  subroutine                *
;*************************************************************
                 ;called from: Main.
Get_Cont_Addr:   ;** gets base segment address for the MBB-80
                 ;** controller from the user at the console.
                 ;** parm in - none
                 ;** parm out - none, updates MB_contbase
            mov  DX,offset msg_getaddr ;addr of get cont msg
            call Print_String       ;write msg to console
        ;get base address keyed in by the user
            mov  CL,Bdos_cnbuf  ;input console string func#
            mov  BX,offset cons_buff ;area for cons input
            mov  byte ptr [BX],cnbuf_size ;tell Bdos size
            mov  DX,BX              ;load parm for Bdos call
            call Bdos               ;read from console
            call Crlf               ;skip a line after input
        ;make sure only four digits keyed in
            mov  BX,offset cons_buff+1 ;byte 1 tells how many
            cmp  byte ptr [BX],4    ;see if exactly four read
            jne  Error_input        ;if not 4, error
        ;make sure all four digits are valid hex
            mov  BX,offset cons_buff+2 ;byte 2 starts data
            xor  AX,AX              ;used for Ascii table index
            mov  CX,4               ;number of digits to check
      Check_valid:
            mov  AL,[BX]            ;move digit to AL for chking
            cmp  AL,030H            ;check to see if too low
            jb   Error_input
            cmp  AL,046H            ;check to see if too high
            ja   Error_input
            cmp  AL,039H            ;chk mid-invalid (3aH-40H)
```

```
            jbe   Valid_hex
            cmp   AL,046H
            jae   Valid_hex
            jmps  Error_input          ;it is in the middle - error
      Valid_hex:
            sub   AX,030H              ;-30H to get table index
            push  BX                   ;save buffer addr
            mov   BX,AX                ;AX is index to table
            mov   AL,Ascii_table[BX]   ;table look up
            pop   BX                   ;restore buffer addr
            mov   byte ptr[BX],AL      ;store hex back in buffer
            inc   BX                   ;next digit
            loop  Check_valid          ;go check it
        ;convert 4 valid hex digits to a binary number in AX
            mov   BX,offset ccns_buff+2 ;byte 2 starts data
            mov   AH,[BX]              ;get first digit
            mov   CL,4                 ;shift it to high nibble
            shl   AH,CL
            inc   BX                   ;increment index
            or    AH,[BX]              ;2nd dig or'ed intc low nibb
            inc   BX                   ;increment index
            mov   AL,[BX]              ;get third digit
            mov   CL,4                 ;shift it to high nibble
            shl   AL,CL
            inc   BX                   ;increment index
            or    AL,[BX]              ;4th dig or'ed into low nibb
        ;store controller base address that was built in AX
            mov   MB_contbase,AX
            jmps  Get_cont_ret         ;go return
        ;error in input, issue message, retry
      Error_input:
            mov   DX,offset msg_errinp ;addr of error message
            call  Print_String         ;write msg to console
            call  Crlf                 ;skip a line
            jmps  Get_Cont_Addr        ;go ask again
      Get_cont_ret:
            ret
;
;*****************************************************************
;*                  GET_TEST_BUFFER  subroutine                  *
;*****************************************************************
                  ;called frcm: Main.
Get_Test_Buffer:  ;** increments pattern and loads test buffer
                  ;** parm in - none
                  ;** parm out - none, effects global vars
            inc   pattern              ;add one (1) to pattern
            mov   AL,pattern           ;pattern to accum for manipul
            mov   CX,MB_buflen         ;loop counter - size of buff
            mov   BX,offset test_buffer ;set index into buffer
      Fill: mov   [BX],AL              ;load a byte
            inc   BX                   ;bump index
            loop  Fill                 ;dec CX, loop if nct zero
            ret
;
;
;*****************************************************************
;*                  HEX_TO_ASCII  subroutine                     *
;*****************************************************************
                  ;called from: End_Pass,Err_Out,Error_Summary.
Hex_Tc_Ascii:     ;** converts a hex number to its hex Ascii
                  ;** parm in - AL has hex byte to ccnvert
                  ;** parm out - DX contains hi&lo Ascii bytes
        ;convert low nibble of AL to Ascii hex digit
            mov   AH,AL                ;save hex # for hi nibble
            and   AL,0fH               ;clear hi 4 bits lc nibble
            add   AL,90H               ;handles 0-9 (90H+40H=130H)
            daa                        ;decimal adjust
            adc   AL,40H               ;handle a-fH (41H-46H Ascii)
```

```
            daa                         ;decimal adjust
            mov  DL,AL                  ;low nibble Ascii for ret
        ;convert high nibble of AL to Ascii hex digit
            mov  AL,AH                  ;move to AL for daa ops
            mov  CL,4                   ;set count for shr 4
            shr  AL,CL                  ;shift hi nibble to lo nibble
            add  AL,90H                 ;handles 0-9 (90H+40H=130H)
            daa                         ;decimal adjust
            adc  AL,40H                 ;handle a-fH (41H-46H Ascii)
            daa                         ;decimal adjust
            mov  DH,AL                  ;high nibble Ascii for ret
            ret
;
;*****************************************************
;*                 INIT_CONT  subroutine             *
;*****************************************************
                ;called from: Main.
Init_Cont:      ;** inits the MBE controller and each device
                ;** parm in - none
                ;** parm out - none
            mov  DX,offset msg_initbegin ;begin init msg addr
            call Print_String       ;write msg to console
        ;initialize page size and mincr loop size
            mov  AX,MB_contbase     ;address of controller base
            mov  ES,AX              ;load ES to address bubble
            mov  AX,MB_maxpages     ;pages per bubble device
            mov  ES:P_loopsize_lo,AL ;loopsize low byte
            mov  ES:P_loopsize_hi,AH ;loopsize hi byte
            mov  ES:P_pagesize_reg,MB_pagesize;page size reg
        ;issue reset command to the controller
            mov  AL,MB_reset_cmd    ;reset mask byte
            mov  ES:P_cmnd_reg,AL   ;issue reset command
        ;initialize each bubble device
            mov  CX,MB_maxdevs+1    ;count for loop-# of devices
            mov  AL,0               ;device # to initialize
  For_each:
            mov  ES:P_select_bubdev,AL ;select each device
            mov  ES:P_cmnd_reg,MB_init_cmd ;init this device
            push AX!push CX!push ES ;save bubble #,counter,ES
            call Wait               ;wait for controller to work
            pop  ES! pop CX! pop AX ;restore ES,cntr,bubble #
            inc  AL                 ;next device number
            loop For_each           ;dec CX, loop if not zero
        ;issue msgs indicating init done and test in progress
            mov  DX,offset msg_initend ;init done message addr
            call Print_String       ;write msg to console
            call Crlf               ;skip an extra line
            mov  DX,offset msg_testing ;testing message addr
            call Print_String       ;write msg to console
            ret
;
;*****************************************************
;*                 LOG_ERROR  subroutine             *
;*****************************************************
                ;called from: Check_Errors.
Log_Error:      ;** log the error for use in pass printout
                ;** parm in - none
                ;** parm out - none, effects global vars
            mov  BX,errptr          ;addr of error log to BX
            inc  byte ptr [BX]      ;add one to error count
            jnz  done_log           ;if not overflow, all done
            dec  byte ptr [BX]      ;inc too big, reduce to max
  done_log:
            ret
;
```

```
;*****************************************************
;*                PRINT_STRING  subroutine           *
;*****************************************************
                ;called from: Close_Up,End_Pass,Err_Out,
                ;             Error_Summary,Get_Cont_Addr,
                ;             Init_Cont,Main,Set_Up.
Print_String:   ;** prints buffer addressed until '$' hit
                ;** parm in - address of buffer in DX
                ;** parm out - none
          mov  CL,Bdos_pstring ;function # for Bdos call
          call Bdos            ;call Bdos and print
          call Crlf            ;skip a line
          ret
;
;*****************************************************
;*                PUTCHAR  subroutine                *
;*****************************************************
                ;called from: Crlf.
Putchar:        ;** writes character from AL to console
                ;** parm in - output char in AL
                ;** parm out - none
          mov  CL,Bdos_conout  ;function # for Bdos call
          mov  DL,AL           ;load char to Bdos reg
          call Bdos            ;call Bdos and send
          ret
;
;*****************************************************
;*                READ_PAGE  subroutine              *
;*****************************************************
                ;called from: Main.
Read_Page:      ;** reads a page into test buffer from bubble
                ;** parm in - none
                ;** parm out - none, effects global vars
      ;select page number
          mov  AX,MB_contbase  ;address of controller base
          mov  ES,AX           ;load ES to address bubble
          mov  AX,curr_page_no ;current page number testing
          mov  ES:P_pagesel_lo,AL ;page select lo byte
          mov  ES:P_pagesel_hi,AH ;page select hi byte
      ;select bubble device and issue read command
          mov  AL,curr_bub_no  ;curr bubble number testing
          mov  ES:P_select_bubdev,AL ;select current dev #
          mov  ES:P_cmnd_reg,MB_read_cmd ;issue read FIFO
          push ES              ;save ES
          call Wait            ;wait for controller to work
          pop  ES              ;restore ES
      ;read from MBB FIFO buffer into test buffer
          mov  CX,MB_buflen    ;count for loop-buffer size
          mov  BX,offset test_buffer ;set index into buffer
  Read_byte:
          mov  AL,ES:P_rdata_reg ;read a byte into accum
          mov  [BX],AL         ;load accum into buffer
          inc  BX              ;increment index
          loop Read_byte       ;dec CX, loop if not zero
          ret
;
;*****************************************************
;*                SET_UP  subroutine                 *
;*****************************************************
                ;called from: Main.
Set_Up:         ;** inits variables and issues signon msg
                ;** parm in - none
                ;** parm out - none, effects global vars
          call Crlf            ;skip an extra line
          call Crlf            ;skip an extra line
          mov  DX,offset msg_signon ;signon msg address
          call Print_String    ;write msg to console
```

```
        mov  DX,offset msg_version ;version msg address
        call Print_String     ;write msg to conscle
        call Crlf             ;skip an extra line
    ;initialize all variables and flags
        mov  newpass_flag,1   ;flag indicating new pass
        mov  curr_bub_no,0    ;current bubble # to 0
        mov  curr_page_nc,0   ;current page # to 0
        mov  pattern,1        ;initial test pattern is 1
        mov  pass_no,1        ;initial pass # is 1
        mov  errptr,offset errlog ;addr of error log
        ret
;
;*****************************************************
;*                WAIT subroutine                    *
;*****************************************************
                ;called from: Init_Cont,Read_Page,Write_Page.
Wait:           ;** checks status of MBB controller for busy
                ;** keeps checking (wait) until not busy
                ;** parm in - none
                ;** parm out - none
        mov  AX,MB_contbase   ;address of controller base
        mov  ES,AX            ;load ES to address bubble
  See_zero:
        mov  AL,ES:P_status_reg ;get status register
        and  AL,MB_busy_check ;is it all zeros ?
        jz   See_zero         ;if so,keep checking for one
  Cont_busy:
        mov  AL,ES:P_status_reg ;get status register
        and  AL,MB_busy_check ;see if busy, and to mask
        jnz  Cont_busy        ;if busy, check again
        ret
;
;*****************************************************
;*              WRITE PAGE  subroutine               *
;*****************************************************
                ;called from: Main.
Write_Page:     ;** writes a page from test_buffer to bubble
                ;** parm in - none
                ;** parm out - none
    ;select page number
        mov  AX,MB_contbase   ;address of controller base
        mov  ES,AX            ;load ES to address bubble
        mov  AX,curr_page_no  ;current page # testing
        mov  ES:P_pagesel_lo,AL ;page select lo byte
        mov  ES:P_pagesel_hi,AH ;page select hi byte
    ;write from test buffer into the MBB FIFO buffer
        mov  CX,MB_buflen     ;count for loop-buffer size
        mov  BX,offset test_buffer ;set index into buffer
  Write_byte:
        mov  AL,[BX]          ;byte from buffer tc accum
        mov  ES:P_wdata_reg,AL ;write a byte to MBB FIFO
        inc  BX               ;increment index
        loop Write_byte       ;dec CX, loop if nct zero
    ;select bubble number and write FIFO buffer to bubble
        mov  AL,curr_bub_no   ;load accum w/ bub#
        mov  ES:P_select_bubdev,AL ;load bubble device #
        mov  ES:P_cmnd_reg,MB_write_cmd ;issue write FIFO
        call Wait             ;wait for controller to work
        ret
;
;*****************************************************
;*                  DATA SEGMENT AREA                *
;*****************************************************
;
    DSEG
```

```
                org 0100H       ;leave room for base page
;
;** -------------------Variables----------------------------*
;
Ascii_table     db  00H,01H,02H,03H,04H,05H,06H,07H,08H,09H
                rb  7   ;for Ascii 3aH to 40H - invalid
                db  0aH,0bH,0cH,0dH,0eH,0fH
cons_buff       rb  conbuf_size ;area for cons string input
curr_bub_no     rb  1           ;bubble device # 0-7 testing
curr_page_no    rw  1           ;bubble page number testing
errlog          rb  MB_maxdevs+1 ;table for dev error count
errptr          rw  1           ;pointer to errlog - index
MB_contbase     dw  0000H       ;base segment addr for MBB-80
newpass_flag    rb  1           ;flag for indicating new pass
pass_no         rb  1           ;pass number
pattern         rb  1           ;test pattern
test_buffer     rb  MB_buflen   ;buffer to hold test data
;
;**-------- string data area for console messages ---------*
;
msg_counts      rb  ((MB_maxdevs+1)*3)
                db  '$'
msg_donebub     db  '  Done with a bubble.$'
msg_donepass    db  'Done with PASS '
  msg_d_pass    rb  2
                db  '$'
msg_endtest     db  '*User terminates testing...'
                db  'returning to CP/M!$'
msg_err         db  '
  msg_e_dev     rb  2
                db  '    '
  msg_e_page    rb  4
                db  '   '
  msg_e_byte    rb  2
                db  '     '
  msg_e_wrote   rb  2
                db  '    '
  msg_e_read    rb  2
                db  '$'
msg_errinp      db  '**ERRCR: not exactly 4 digits entered,'
                db  ' or invalid hex digits!!$'
msg_getaddr     db  cr,lf,'Key in 4 digit segment base addr'
                db  'ess for MBB-80 controller.',cr,lf
                db  'Must be in hex (4 digits, then CR only)'
                db  ' => $'
msg_header      db  'Bubble  Page  Byte  Wrote  Read$'
msg_initbegin   db  'Initializing the controller...$'
msg_initend     db  'Controller is initialized.$'
msg_signon      db  '
                db  '** MBB-80 CP/M-86 DIAGNOSTIC TEST **$'
msg_summary     db  'Total errors for each device (0-7):$'
msg_testing     db  'Testing...Hit any char (& CR!) '
                db  'to stop after this pass.$'
msg_version     db  '
                db  'Single-Page Mode   Version 1.0$'
                db  0           ;GENCMD to fill last address
;
;*************** end of variables ***************************
;
        ESEG
;
;************************************************************
;*                MBB-80 CONTROLLER AND PORTS               *
;************************************************************
;
P_pagesel_lo    rb  1           ;ls byte for page select, (0)
P_pagesel_hi    rb  1           ;ms 2 bits for page select,(1)
```

```
P_cmnd_reg      rb  1           ;command register, (2)
P_rdata_reg     rb  1           ;read data register, (3)
P_wdata_reg     rb  1           ;write data register, (4)
P_status_reg    rb  1           ;status register, (5)
P_pagecnt_lo    rb  1           ;ls byte for page counter,(6)
P_pagecnt_hi    rb  1           ;ms 2 bits for page cntr,(7)
P_loopsize_lo   rb  1           ;ls byte for minor loop sz,(8)
P_loopsize_hi   rb  1           ;ms 2 bits for min loop sz,(9)
                rw  1           ;internal use(page pos), (A,B)
P_pagesize_reg  rb  1           ;page size register, (C)
                rw  1           ;TI use only, (D,E)
P_select_bubdev rb  1           ;two uses: sel bubble dev (F)
P_int_flag      equ P_select_bubdev ;    interrupt flag (F)
;
;********* end of Controller and Port definitions *********
;
;
;*************************************************************
;*                  End of Program DIAG86S                   *
;*************************************************************
                END
```

APPENDIX C

PROGRAM LISTING OF DIAG86M.A86

```
;
;  FILENAMES: Pascal = MB.DIAG86M.TEXT
;             CP/M = DIAG86M.CMD
;
;*************************************************************
;*  8086 DIAGNOSTIC TEST FOR PC/M MBB-80 BUBBLE MEMORIES     *
;*************************************************************
;
;  CONFIGURATION:
;      HOST - Intel 86/12A SBC, 20 address lines, MDS system,
;             Data bus on 86/12A converting to low 8 bits
;             all high.
;      MBB - Interrupts enabled if using vectored interrupts.
;            Interrupts disabled by disconnecting the inter-
;            rupt jumper on the MBB board if not vectoring
;            interrupts. Multi-page mode, 20 address lines.
;
;  This program writes and then reads a test pattern in each
;  sector of each bubble chip on MBB-80 boards. Error
;  diagnostics are printed as errors are found. An error log
;  is printed at the end of each pass.  Testing is continuous
;  until any character is keyed into the console.
;
;  The MBB-80 controller base address is read into variable
;  'MB_contbase'. MBB-80 address select pins must correspond
;  to this address. This program uses memory mapped I/O
;  through the base address.
;
;*************************************************************
;*  Jeffrey Neufeld and Michael Hicklin,  CS-03,  Thesis     *
;*************************************************************
;
;* Bdos function numbers for calls *
Bdos_conbuf     equ 10      ;console input string function #
Bdos_conout     equ 2       ;console output char function #
Bdos_constat    equ 11      ;get console status function #
Bdos_pstring    equ 9       ;print string until '$' function #
Bdos_reset      equ 0       ;CP/M-86 reset to CCP function #
;
;* 8259a PIC port assignments
PICp0           equ 0c0H    ;8259a port 0
PICp1           equ 0c2H    ;8259a port 1
;
;* MBB characteristics *
MB_buflen       equ 144     ;buffer length for sector
MB_int_mask     equ 11111101B ;mask to enable MBB interrupt
MB_int_type     equ 17      ;type 16 is IR0 as defined to
                            ;8259a PIC in ROM init. MBB will
                            ;generate interrupts over this
                            ;type.
MB_maxdevs      equ 7       ;bubble devices are #0-#7
MB_maxpages     equ 641     ;# of pages on each bubble device
MB_maxsectors   equ 80      ;# of log sectors on each bub dev
MB_pages_sec    equ 8       ;# of pages per logical sector
MB_pagesize     equ 18      ;bubble device page size
MB_skew         equ 12      ;skew for page translation
;
```

```
;* MBB command masks and status masks *
MB_busy_check   equ 00100000B ;cont busy? status check (20H)
MB_init_cmd     equ 00000001B ;init the controller (01H)
MB_int_inhibit  equ 10000000B ;int inhibit/reset mask (80H)
MB_chkint_mask  equ 10000000B ;mask testing if int set (80H)
MB_multi_page   equ 00010000B ;multi-page mode command (10H)
MB_read_cmd     equ 00010010B ;multi-page read command (12H)
MB_reset_cmd    equ 01000000B ;reset the controller (40H)
MB_write_cmd    equ 00010100B ;multi-page write command (14H)
;
;* Miscellaneous equates *
blank           equ 020H    ;Ascii blank
conbuf_size     equ 80      ;size for input buffer for console
cr              equ 0dH     ;Ascii carriage return cont char
true            equ -1      ;for conditional assembly
false           equ not true ;for conditional assembly
lf              equ 0aH     ;Ascii line feed control char
vectored_int    equ false   ;this controls the assembly.
                            ;true=use hard interrupt to CPU.
                            ;false=poll int reg on MBB.
;
;
;*****************************************************
;*                MAIN PROGRAM - DRIVER              *
;*****************************************************
;
        CSEG
;
DIAG86M:  call Set_Up           ;dc initialization
          call Get_Cont_Addr    ;get base address for MBB-80
          call Init_Cont        ;init the cont and devices
  Test_loop:
          call Get_Test_Buffer ;get test pattern, fill buff
          call Write_Sector     ;write a sector to bubble
          call Read_Sector      ;read a sector from bubble
          call Check_Errors     ;check errors in write/read
       ;advance to next sector in device,see if last sector
          inc  curr_sector_no   ;increment current sector #
          cmp  curr_sector_no,MB_maxsectors ;last sector ?
          jnz  Test_loop        ;if not, test next sector
       ;was last sector,advance to next bub dev on board
          mov  DX,offset msg_donebub ;addr of done bub msg
          call Print_String     ;write msg to console
          cmp  curr_bub_no,MB_maxdevs ;last bubble on board?
          jz   Done_pass        ;if so, done with a pass
       ;prepare to test next bubble device
          inc  curr_bub_no      ;if not, increment device #
          mov  curr_sector_no,0 ;set sector # back to zero
          inc  errptr           ;ptr to next entry (dev)
          jmp  Test_loop        ;go test next device
       ;finished with all devices on board, print summary
       ;prepare to run another pass if not stopped by user
  Done_pass:
          call Error_Summary    ;print error summary
          call End_Pass         ;end of pass housekeeping
       ;see if anything keyed in at the console
          mov  CL,Bdos_constat  ;function # for Bdos call
          call Bdos             ;call Bdos to get cons status
          cmp  AL,01            ;01=char keyed in, 00=nothing
          jz   Done_test        ;something keyed, user quits
       ;user wants to continue
          mov  DX,offset msg_testing ;addr of testing msg
          call Print_String     ;write msg to console
          jmp  Test_loop        ;keep testing
    ;user wanted to quit the testing
  Done_test:
          call Close_Up         ;do end of run housekeeping
          mov  CL,Bdos_reset    ;function # for Bdos call
```

```
          mov  DL,0              ;parameter to release memory
          call Bdos              ;call Bdos to terminate prog
;
;***************** end of Main Program ********************
;
;
;*************************************************************
;*                BDOS (CP/M-86)  subroutine                 *
;*************************************************************
                ;called from: Close_Up, Get_Cont_Addr, Main,
                ;             Print_String, Putchar.
Bdcs:           ;** entry to Bdos via software interrupt 224
                ;** parm in - caller loads regs as per req
                ;** parm out - as supplied by Bdos returns
          int  224              ;8086 software interrupt
          ret
;
;*************************************************************
;*                CHECK ERRORS  subroutine                   *
;*************************************************************
                ;called from: Main.
Check_Errors:   ;** see if read what was written
                ;** parm in - none
                ;** parm out - none
          mov  AL,pattern       ;pattern to accum for manipul
          mov  CX,NB_buflen     ;counter for loop thru buffer
          mov  BX,offset test_buffer ;index into test buffer
  Test_byte:
          cmp  [BX],AL          ;compare buff to pattern
          jz   Good_test        ;if good, check next byte
          push AX!push BX!push CX ;save patt/buff addr/cntr
          call Err_Out          ;it is bad, print error
          call Log_Error        ;           log error
          pop  CX!pop BX!pop AX ;restore cntr/buff addr/patt
  Good_test:
          inc  BX               ;increment index
          loop Test_byte        ;dec CX and loop if not zero
          ret
;
;*************************************************************
;*                CLOSE UP  subroutine                       *
;*************************************************************
                ;called from: Main.
Close_Up:       ;** reads garbage from console,issues goodbye
                ;** parm in - none
                ;** parm out - none
       ;clear stop input characters from the console buffer
          mov  CL,Bdos_conbuf  ;input console string func#
          mov  BX,offset cons_buff;area for cons input
          mov  byte ptr [BX],conbuf_size;tell Bdos buff size
          mov  DX,BX            ;load parameter reg for Bdos
          call Bdos             ;read the console
       ;issue goodbye message
          call Crlf             ;skip extra line
          mov  DX,offset msg_endtest ;addr of end test msg
          call Print_String     ;write msg to console
          ret
;
;
```

```
;*************************************************************
;*                  COMPUTE_PAGENO  subroutine               *
;*************************************************************
                ;called from: Read_Sector, Write_Sector.
Compute_Pageno: ;** computes 1st page # for a given sector
                ;** parm in - none, works on curr_sector_no
                ;** parm out - none, updates curr_page_no
          xor   AX,AX             ;set AX to zero
          cmp   AL,curr_sector_no ;is it sector 0 ?
          jz    Store_page        ;if so, no translation
          xor   CX,CX             ;clear CX for counter
          mov   CL,curr_sector_no ;cntr for translate loop
  Add_skew:
          add   AX,MB_skew        ;# of pages between sectors
          clc                     ;clear carry
          sbb   AX,MB_maxpages    ;mod to # of pages
          jae   Dec_sector        ;jump if positive (CF=0)
          add   AX,MB_maxpages    ;went neg, add back # pages
  Dec_sector:
          loop Add_skew           ;dec sector #, add skew again
  Store_page:
          mov   curr_page_no,AX ;store page number
          ret
;
;
;*************************************************************
;*                  CRLF  subroutine                         *
;*************************************************************
                ;called from: Close_Up, Get_Cont_Addr,
                ;End_Pass,Init_Cont,Main,Print_String,Set_Up.
Crlf:           ;** sends carriage return,line feed to cons
                ;** parm in - none
                ;** parm out - none
          mov   AL,cr             ;carriage return char
          call  Putchar           ;write it to console
          mov   AL,lf             ;line feed char
          call  Putchar           ;write it to console
          ret
;
;
;*************************************************************
;*                  END_PASS  subroutine                     *
;*************************************************************
                ;called from: Main.
End_Pass:       ;** performs end of pass housekeeping
                ;** parm in - none
                ;** parm out - none, effects global vars
      ;convert pass # to Ascii and print after pass message
          mov   AL,pass_no        ;pass number to accum
          call  Hex_To_Ascii      ;convert to Ascii
          mov   BX,offset msg_d_pass;addr of pass # in msg
          mov   byte ptr [BX],DH ;load high byte to msg
          inc   BX                ;bump to next position in msg
          mov   byte ptr [BX],DL ;load low byte to msg
          mov   DX,offset msg_donepass ;addr of done pass msg
          call  Print_String      ;write msg to console
          call  Crlf              ;skip a line
      ;inc pass number and reset all variables for new pass
          inc   pass_no           ;add one to pass number
          mov   newpass_flag,1    ;set new-pass flag on
          mov   curr_bub_no,0     ;reset to bubble device 0
          mov   curr_sector_no,0 ;reset sector number to 0
          mov   errptr,offset errlog ;reset addr of error log
          ret
;
;
```

```
;*************************************************************
;*                  ERR_OUT  subroutine                      *
;*************************************************************
                ;called from: Check_Errors.
Err_Out:        ;** issue an error message to the console
                ;** parm in - BX addr in buff of byte error
                ;** parm out - none, effects global vars
          push BX ! push BX     ;save addr of error twice
          cmp  newpass_flag,1   ;is this a new pass ?
          jnz  Prt_err          ;if not, print error now
          mov  newpass_flag,0   ;turn flag off
          mov  DX,offset msg_header ;load addr of header
          call Print_String     ;print the header
      ;put zeros into all error counts in the log
          mov  CX,NB_maxdevs+1 ;count for # of dev to loop
          mov  BX,offset errlog ;addr of error log
  Clr_log:
          mov  byte ptr [BX],0 ;clear log entry error count
          inc  BX               ;bump pointer to next entry
          loop Clr_log          ;dec CX and loop if not zero
  Prt_err:
          mov  AL,curr_bub_no   ;bub dev # to accum
          call Hex_To_Ascii     ;convert to Ascii
          mov  msg_e_dev,DH     ;move in high byte to msg
          mov  msg_e_dev+1,DL   ;move in low byte to msg
      ;load page number of error
          mov  AL,byte ptr curr_page_no+1;hi byte of page#
          call Hex_To_Ascii     ;convert to Ascii
          mov  msg_e_page,DH    ;high byte to msg(dig 1)
          mov  msg_e_page+1,DL ;low byte to msg(dig 1)
          mov  AL,byte ptr curr_page_no ;lo byte of page#
          call Hex_To_Ascii     ;convert to Ascii
          mov  msg_e_page+2,DH ;high byte to msg(dig 2)
          mov  msg_e_page+3,DL ;low byte to msg(dig 2)
      ;compute and load byte offset of error in page
          pop  BX               ;restore addr err byte offset
          addr_buff equ offset test_buffer ;for computation
          sub  BX,addr_buff     ;compute err offset in buff
          mov  AL,BL            ;offset to AL for conversion
          call Hex_To_Ascii     ;convert to Ascii
          mov  msg_e_byte,DH    ;move in high byte to msg
          mov  msg_e_byte+1,DL ;move in low byte to msg
      ;load pattern that was written and what was read back
          mov  AL,pattern       ;load pattern just written
          call Hex_To_Ascii     ;convert to Ascii
          mov  msg_e_wrote,DH   ;move in high byte to msg
          mov  msg_e_wrote+1,DL ;move in low byte to msg
          pop  BX               ;restore addr of err offset
          mov  AL,[BX]          ;load byte just read back
          call Hex_To_Ascii     ;convert to Ascii
          mov  msg_e_read,DH    ;move in high byte to msg
          mov  msg_e_read+1,DL ;move in low byte to msg
          mov  DX, offset msg_err ;addr of total error msg
          call Print_String     ;print the error message
          ret
;
;*************************************************************
;*                  ERROR SUMMARY  subroutine                *
;*************************************************************
                ;called from: Main.
Error_Summary:  ;** outputs summary of errors on each device
                ;** parm in - none
                ;** parm out - none
          mov  DX,offset msg_summary ;addr of summary msg
          call Print_String     ;write msg to console
      ;step thru errlog-convert to Ascii - print err counts
          mov  CX,NB_maxdevs+1 ;count for loop - # of devs
```

```
            mov  BX,offset errlog ;addr of error log
            mov  DI,offset msg_counts ;addr of msg sum counts
   prt_loop:
            mov  AL,[BX]          ;get count from errcr log
            push BX!push CX!push DI ;save addr,counter,index
            call Hex_To_Ascii     ;convert to Ascii
            pop  DI! pop CX! pop BX ;rest index,counter,addr
            mov  byte ptr [DI],DH ;load high byte tc msg
            inc  DI               ;bump to next pos in msg
            mov  byte ptr [DI],DL ;load low byte to msg
            inc  DI               ;bump to next pos in msg
            mov  byte ptr [DI],blank ;Ascii blank to msg
            inc  DI               ;bump to next pos in msg
            inc  BX               ;increment buff addr to next
            loop prt_loop         ;dec CX and loop if not zero
            mov  DX,offset msg_ccunts ;addr of msg sum counts
            call Print_String     ;write msg to console
            ret
;
;
;*****************************************************
;*              GET CCNT ADDR  subroutine            *
;*****************************************************
                  ;called from: Main.
Get_Cont_Addr:    ;** gets base segment address for the MBB-80
                  ;** ccntroller from the user at the console.
                  ;** parm in - none
                  ;** parm out - ncne, updates MB_contbase
            mov  DX,offset msg_getaddr ;addr of get cont msg
            call Print_String     ;write msg to conscle
         ;get base address keyed in by the user
            mov  CL,Bdos_conbuf   ;input console string func#
            mov  BX,offset ccns_buff ;area for cons input
            mov  byte ptr[BX],conbuf_size ;tell Bdos size
            mov  DX,BX            ;load parm for Bdos call
            call Bdos             ;read from console
            call Crlf             ;skip a line after input
         ;make sure only four digits keyed in
            mov  BX,offset ccns_buff+1 ;byte 1 tells how many
            cmp  byte ptr[BX],4   ;see if exactly fcur read
            jne  Error_input      ;if nct 4, error
         ;make sure all fcur digits are valid hex
            mov  BX,offset ccns_buff+2 ;byte 2 starts data
            xor  AX,AX            ;used for Ascii table index
            mov  CX,4             ;number of digits to check
      Check_valid:
            mov  AL,[BX]          ;move digit to AL for chking
            cmp  AL,030H          ;check to see if tco low
            jb   Error_input
            cmp  AL,046H          ;check to see if too high
            ja   Error_input
            cmp  AL,039H          ;chk mid-invalid (3aH-40H)
            jbe  Valid_hex
            cmp  AL,047H
            jae  Valid_hex
            jmps Error_input      ;it is in the middle - error
      Valid_hex:
            sub  AX,030H          ;-30H to get table index
            push BX               ;save buffer addr
            mov  BX,AX            ;AX is index to table
            mov  AL,Ascii_table[BX] ;table look up
            pop  BX               ;restore buffer addr
            mov  byte ptr[BX],AL  ;store hex back in buffer
            inc  BX               ;next digit
            loop Check_valid      ;go check it
         ;convert 4 valid hex digits to a binary number in AX
            mov  BX,offset ccns_buff+2 ;byte 2 starts data
            mov  AH,[BX]          ;get first digit
```

```
            mov  CL,4               ;shift it to high nibble
            shl  AH,CL
            inc  BX                 ;increment index
            or   AH,[BX]            ;2nd dig or'ed intc low nibb
            inc  BX                 ;increment index
            mov  AL,[BX]            ;get third digit
            mov  CL,4               ;shift it to high nibble
            shl  AL,CL
            inc  BX                 ;increment index
            or   AL,[BX]            ;4th dig or'ed into low nibb
          ;store controller base address that was built in AX
            mov  MB_contbase,AX
            jmps Get_cont_ret       ;go return
          ;error in input, issue message, retry
        Error_input:
            mov  DX,offset msg_errinp ;addr of error message
            call Print_String       ;write msg to conscle
            call Crlf               ;skip a line
            jmps Get_Cont_Addr      ;go ask again
        Get_cont_ret:
            ret
;
;
;*****************************************************************
;*                  GET_TEST_BUFFER  subrcutine                  *
;*****************************************************************
                    ;called frcm: Main.
Get_Test_Buffer:    ;** increments pattern and loads test buffer
                    ;** parm in - none
                    ;** parm out - none, effects global vars
            inc  pattern            ;add one (1) to pattern
            mov  AL,pattern         ;pattern to accum for manipul
            mov  CX,MB_buflen       ;loop counter - size of buff
            mov  BX,offset test_buffer ;set index into buffer
            mov  [BX],AL            ; load a byte
            inc  BX                 ;bump index
            loop Fill               ;dec CX, loop if not zero
            ret
;
;
;*****************************************************************
;*                  HEX_TO_ASCII  subroutine                     *
;*****************************************************************
                    ;called from: End_Pass,Err_Out,Error_Summary.
Hex_To_Ascii:       ;** converts a hex number to its hex Ascii
                    ;** parm in - AL has hex byte to ccnvert
                    ;** parm cut - DX contains hi&lo Ascii bytes
        ;convert low nibble cf AL to Ascii hex digit
            mov  AH,AL              ;save hex # for hi nibble
            and  AL,0fH             ;clear high 4 bits lo nibble
            add  AL,90H             ;handles 0-9 (90H+40H=130H)
            daa                     ;decimal adjust
            adc  AL,40H             ;handle a-fH (41H-46H Ascii)
            daa                     ;decimal adjust
            mov  DL,AL              ;lcw nibble Ascii fcr ret
        ;convert high nibble cf AL to Ascii hex digit
            mov  AL,AH              ;mcve to AL for daa ops
            mov  CL,4               ;set count for shr 4
            shr  AL,CL              ;shift hi nibble tc lo nibble
            add  AL,90H             ;handles 0-9 (90H+40H=130H)
            daa                     ;decimal adjust
            adc  AL,40H             ;handle a-fH (41H-46H Ascii)
            daa                     ;decimal adjust
            mov  DH,AL              ;high nibble Ascii for ret
            ret
;
;
```

```
;*****************************************************
;*                INIT_CONT  subroutine              *
;*****************************************************
                ;called from: Main.
Init_Cont:      ;** inits the MBE controller and each device
                ;** parm in - none
                ;** parm out - none
            mov  DX,offset msg_initbegin ;begin init msg addr
            call Print_String    ;write msg to console
        ;initialize page size and mincr loop size
            mov  AX,MB_contbase  ;address of controller base
            mov  ES,AX           ;load ES to address bubble
            mov  AX,MB_maxpages  ;pages per bubble device
            mov  ES:P_loopsize_lo,AL ;loopsize low byte
            mov  ES:P_loopsize_hi,AH ;loopsize hi byte
            mov  ES:P_pagesize_reg,MB_pagesize;page size reg
        ;issue reset command to the controller
            mov  AL,MB_reset_cmd ;reset mask byte
            mov  ES:P_cmnd_reg,AL ;issue reset command
        ;initialize each bubble device
            mov  CX,MB_maxdevs+1 ;count for loop-# of devices
            mov  AL,0            ;device # to initialize
  For_each:
            mov  ES:P_select_bubdev,AL ;select each device
            mov  ES:P_cmnd_reg,MB_init_cmd ;init this device
            push AX!push CX!push ES ;save bubble #,counter,ES
            call Wait            ;wait for controller to work
            pop  ES! pop CX! pop AX ;restore ES,cntr,bubble #
            inc  AL              ;next device number
            loop For_each        ;dec CX, loop if not zero
        ;issue msgs indicating init done and test in progress
            mov  DX,offset msg_initend ;init done message addr
            call Print_String    ;write msg to console
            call Crlf            ;skip an extra line
            mov  DX,offset msg_testing ;testing message addr
            call Print_String    ;write msg to console
            ret
;
;*****************************************************
;*                 LOG_ERROR  subroutine             *
;*****************************************************
                ;called from: Check_Errors.
Log_Error:      ;** log the error for use in pass printout
                ;** parm in - none
                ;** parm out - none, effects global vars
            mov  BX,errptr       ;addr of error log to BX
            inc  byte ptr [BX]   ;add one to error count
            jnz  done_log        ;if not overflow, all done
            dec  byte ptr [BX]   ;inc too big, reduce to max
  done_log:
            ret
;
;*****************************************************
;*                PRINT_STRING  subroutine           *
;*****************************************************
                ;called from: Close_Up, End_Pass, Err_Out,
                ; Error_Summary, Get_Cont_Addr, Init_Cont,
                ; Main, Set_Up.
Print_String:   ;** prints buffer addressed until '$' hit
                ;** parm in - address of buffer in DX
                ;** parm out - none
            mov  CL,Bdos_pstring ;function # for Bdos call
            call Bdos            ;call Bdos and print
            call Crlf            ;skip a line
            ret
;
```

```
;
;*****************************************************
;*                PUTCHAR  subroutine                *
;*****************************************************
                ;called from: Crlf.
Putchar:        ;** writes character from AL to console
                ;** parm in - output char in AL
                ;** parm out - none
        mov  CL,Bdos_conout  ;function# for Bdos call
        mov  DL,AL           ;load char to Bdos reg
        call Bdos            ;call Bdos and send
        ret
;
;*****************************************************
;*                READ_SECTOR  subroutine            *
;*****************************************************
                ;called from: Main.
Read_Sector:    ;** reads sector into test buffer from bubble
                ;** parm in - none
                ;** parm out - none, effects global vars
        call Compute_Pageno  ;compute 1st page# of sector
    ;establish addressability to controller
        mov  AX,MB_contbase  ;address of controller base
        mov  ES,AX           ;load ES to address bubble
    ;set multipage mode
        mov  ES:P_cmnd_reg,MB_multi_page ;multipage mode
    ;load first page number for transfer
        mov  AX,curr_page_no ;current page number testing
        mov  ES:P_pagesel_lo,AL ;page select lo byte
        mov  ES:P_pagesel_hi,AH ;page select hi byte
    ;set number of pages to transfer = pages/sector
        mov  ES:P_pagecnt_lo,MB_pages_sec ;# pages to xfer
        mov  ES:P_pagecnt_hi,0 ;hi byte of # is zero
    ;set up buffer to receive data
        mov  CX,MB_buflen    ;count for loop-buffer size
        mov  BX,offset test_buffer ;set index into buffer
    ;select bubble device and issue read command
        mov  AL,curr_bub_no  ;current bubble # testing
        mov  ES:P_select_bubdev,AL ;select current dev #
        mov  ES:P_cmnd_reg,MB_read_cmd ;read from FIFO
    ;wait for interrupt from controller
  Read_int:
      IF  vectored_int
        cmp  interrupt_flag,0 ;will be set by int handler
        jz   Read_int         ;if zero, keep checking
        mov  interrupt_flag,0 ;reset interrupt flag
      ENDIF  ;vectored_int

      IF  not vectored_int
        mov AL,ES:P_int_flag    ;get interrupt status
        and AL,MB_chkint_mask   ;has interrupt been set?
        jz Read_int             ;if not, keep checking
      ENDIF  ;not vectored_int
    ;read from MBB FIFO buffer into test buffer
        mov  AL,ES:P_rdata_reg ;read a byte into accum
        mov  [BX],AL           ;load accum into buffer
        inc  BX                ;increment index
        loop Read_int          ;dec CX, loop if not zero
        push ES                ;save ES
        call Wait              ;wait for controller to stop
        pop  ES                ;restore ES
        mov  ES:P_cmnd_reg,MB_int_inhibit ;clear cont int
        ret
;
;
```

```
;*****************************************************
;*                SET UP  subroutine                 *
;*****************************************************
                ;called from: Main.
Set_Up:         ;** inits variables and issues signon msg
                ;** parm in - none
                ;** parm out - none, effects global vars
          call Crlf             ;skip an extra line
          call Crlf             ;skip an extra line
          mov  DX,offset msg_signon ;signon message address
          call Print_String     ;write msg to console
          mov  DX,offset msg_version ;version msg address
          call Print_String     ;write msg to console
          call Crlf             ;skip an extra line
      ;initialize all variables and flags
          mov  newpass_flag,1   ;set flag indicating new pass
          mov  curr_bub_no,0    ;current bubble # to 0
          mov  curr_sector_no,0 ;current sector # to 0
          mov  pattern,1        ;initial test pattern is 1
          mov  pass_no,1        ;initial pass # is 1
          mov  errptr,offset errlog ;addr of error log
       ;load MB interrupt vector address in CP/M low memory
          push DS               ;save this pgm's DS
          mov  AX,0             ;lowest memory
          mov  DS,AX            ;make it addressable
          mov  MB_int_segment,CS ;int vector CS is pgm CS
          mov  MB_int_offset,offset Trap_Handler;trap handlr
          pop  DS               ;restore this pgm's DS
       ;set up 8259a PIC to recognize interrupt from MBB-80
          mov  AL,MB_int_mask   ;mask to enable MB interrupt
          out  PICp1,AL         ;send mask to 8259a - OWC1
          sti
          ret
;
;*****************************************************
;*              TRAP_HANDLER  subroutine             *
;*****************************************************
                ;called from: Vectored to from CP/M interrupt
Trap_Handler:   ;** sets the interrupt flag semaphore to one
                ;** parm in - none
                ;** parm out - none
          mov  interrupt_flag,1 ;set the interrupt flag on
          iret                  ;return from interrupt
;
;*****************************************************
;*                WAIT subroutine                    *
;*****************************************************
                ;called from: Init_Cont, Read_Sector,
                ;             Write_Sector.
Wait:           ;** checks status of MBB controller for busy
                ;** keeps checking (wait) until not busy
                ;** parm in - none
                ;** parm out - none
          mov  AX,MB_contbase   ;address of controller base
          mov  ES,AX            ;load ES to address bubble
  See_zero:
          mov  AL,ES:P_status_reg ;get status register
          and  AL,MB_busy_check ;is it all zeros ?
          jz   See_zero         ;if so,keep checking for one
  Cont_busy:
          mov  AL,ES:P_status_reg ;get status register
          and  AL,MB_busy_check ;see if busy, and to mask
          jnz  Cont_busy        ;if busy, check again
          ret
;
;
```

```
;*************************************************************
;*                  WRITE SECTOR  subroutine                 *
;*************************************************************
                ;called from: Main.
Write_Sector:   ;** writes sector from test_buffer to bubble
                ;** parm in - none
                ;** parm out - none
        call Compute_Pageno  ;compute 1st page# of sector
      ;establish addressability to controller
        mov  AX,MB_contbase  ;address of controller base
        mov  ES,AX           ;load ES to address bubble
      ;set multipage mode
        mov  ES:P_cmnd_reg,MB_multi_page ;multipage mode
      ;load first page number for transfer
        mov  AX,curr_page_no ;current page number testing
        mov  ES:P_pagesel_lo,AL ;page select lo byte
        mov  ES:P_pagesel_hi,AH ;page select hi byte
      ;set number of pages to transfer = pages/sector
        mov  ES:P_pagecnt_lo,MB_pages_sec ;# pages to xfer
        mov  ES:P_pagecnt_hi,0 ;hi byte of # is zero
      ;set up buffer to send data
        mov  CX,MB_buflen-1  ;count for loop-buffer size
        mov  BX,offset test_buffer ;set index into buffer
      ;select bubble device and issue write cmd
        mov  AL,curr_bub_no  ;current bubble # testing
        mov  ES:P_select_bubdev,AL ;select current dev #
        mov  AL,[BX]         ;load first byte
        mov  ES:P_wdata_reg,AL ;write a byte to FIFO buff
        inc  BX              ;increment index
        mov  ES:P_cmnd_reg,MB_write_cmd ;write FIFO buff
      ;wait for interrupt from controller
  Write_int:

        IF  vectored_int
          cmp  interrupt_flag,0 ;will be set by int handler
          jz   Write_int        ;if zero, keep checking
          mov  interrupt_flag,0 ;reset interrupt flag
        ENDIF  ;vectored_int

        IF  not vectored_int
          mov AL,ES:P_int_flag    ;get interrupt status
          and AL,MB_chkint_mask   ;has interrupt been set?
          jz Write_Int            ;if not, keep checking
        ENDIF  ;not vectored_int

      ;write into MBB FIFO buffer from test buffer
        mov  AL,[BX]         ;byte from buffer to accum
        mov  ES:P_wdata_reg,AL ;write a byte to FIFO buff
        inc  BX              ;increment index
        loop Write_int       ;dec CX, loop if not zero
        push ES              ;save ES
        call Wait            ;wait for controller to stop
        pop  ES              ;restore ES
        mov  ES:P_cmnd_reg,MB_int_inhibit ;clear cont int
        ret
;*************************************************************
;*                  DATA SEGMENT AREA                        *
;*************************************************************
;
      DSEG
                org 0100H     ;leave room for base page
;
;** ------------------Variables--------------------------*
;
Ascii_table     db  00H,01H,02H,03H,04H,05H,06H,07H,08H,09H
                rb  7  ;for Ascii 3aH to 40H - invalid
                db  0aH,0bH,0cH,0dH,0eH,0fH
```

```
cons_buff       rb  conbuf_size ;area for cons string input
curr_bub_no     rb  1          ;bubble device # 0-7 testing
curr_page_no    rw  1          ;bubble page # testing
curr_sector_no  rb  1          ;bubble log sector # testing
errlog          rb  MB_maxdevs+1 ;table for dev error count
errptr          rw  1          ;pointer to errlog - index
interrupt_flag  db  0          ;int flag - semaphore,from MBB
MB_contbase     dw  0000H      ;base segment addr for MBB-80
newpass_flag    rb  1          ;flag for indicating new pass
pass_no         rb  1          ;pass number
pattern         rb  1          ;test pattern
test_buffer     rb  MB_buflen ;buffer to hold test data
;
;**-------- string data area for console messages ---------*
;
msg_ccunts      rb  ((MB_maxdevs+1)*3)
                db  '$'
msg_donebub     db  '  Done with a bubble.$'
msg_donepass    db  'Done with PASS '
  msg_d_pass    rb  2
                db  '$'
msg_endtest     db  '*User terminates testing...'
                db  'returning to CP/M!$'
msg_err         db  '  '
  msg_e_dev     rb  2
                db  '    '
  msg_e_page    rb  4
                db  '   '
  msg_e_byte    rb  2
                db  '     '
  msg_e_wrote   rb  2
                db  '    '
  msg_e_read    rb  2
                db  '$'
msg_errinp      db  '**ERROR: not exactly 4 digits entered,'
                db  ' or invalid hex digits!!$'
msg_getaddr     db  cr,lf,'Key in 4 digit segment base addr'
                db  'ess for MBB-80 controller.',cr,lf
                db  'Must be in hex (4 digits, then CR only)'
                db  ' => $'
msg_header      db  'Bubble  Page  Byte  Wrote  Read$'
msg_initbegin   db  cr,lf,'Initializing the controller...$'
msg_initend     db  'Controller is initialized.$'
msg_signon      db  '                '
                db  '** MBE-80 CP/M-86 DIAGNOSTIC TEST **$'
msg_summary     db  'Total errors for each device (0-7):$'
msg_testing     db  'Testing...Hit any char (& CR!) '
                db  'to stop after this pass.$'
msg_version     db  '                '
                db  'Multi-Page Mode  Version 1.0',cr,lf
                IF vectored_int
                db  '                '
                db  '    Vectored Interrupts$'
                ENDIF ;vectored_int
                IF not vectored_int
                db  '                '
                db  '    Polled  Interrupts$'
                ENDIF ;not vectored_int
                db  0          ;GENCMD to fill last address
;
;**************** end of variables ************************
;
        ESEG
```

```
;*****************************************************
;*                MBB-80 CONTROLLER AND PORTS        *
;*****************************************************
;
P_pagesel_lo    rb  1           ;ls byte for page select, (0)
P_pagesel_hi    rb  1           ;ms 2 bits for page select,(1)
P_cmnd_reg      rb  1           ;command register, (2)
P_rdata_reg     rb  1           ;read data register, (3)
P_wdata_reg     rb  1           ;write data register, (4)
P_status_reg    rb  1           ;status register, (5)
P_pagecnt_lo    rb  1           ;ls byte for page counter, (6)
P_pagecnt_hi    rb  1           ;ms 2 bits for page cnter, (7)
P_lccpsize_lo   rb  1           ;ls byte for minor lcop sz,(8)
P_lcopsize_hi   rb  1           ;ms 2 bits for min lcop sz,(9)
                rw  1           ;internal use(page pcsl, (A,B)
P_pagesize_reg  rb  1           ;page size register, (C)
                rw  1           ;TI use only, (D,E)
P_select_bubdev rb  1           ;two uses: sel bubble dev (F)
P_int_flag      equ P_select_bubdev ;    interrupt flag (F)
;********** end of Controller and Port definitions **********
;
;*****************************************************
;*                  DUMMY DATA SECTION               *
;*****************************************************
;
        DSEG    0               ;absolute low memory
        org     0               ;start CP/M interrupt vectors
                                ;pad to int type for MBB
                rw  2*(MB_int_type)
MB_int_offset   rw  1           ;addr cf int vector cffset
MB_int_segment  rw  1           ;addr of int vector segment
;
;*****************************************************
;*                End of Program DIAG86M             *
;*****************************************************
                END
```

APPENDIX D

PROGRAM LISTING OF MB80FMT.A86

```
;
;  FILENAMES: Pascal = MB.MB80FMT.TEXT
;             CP/M = MB80FMT.CMD
;
;*************************************************************
;*   8086 FORMAT PROGRAM FOR PC/M MBB-80 BUBBLE MEMORIES     *
;*************************************************************
;
;  CONFIGURATION:
;      HOST - Intel 86/12A SBC, 20 address lines, MDS system,
;             Data bus on 86/12A converting to low 8 bits
;             all high.
;      MBB - Interrupts disabled by disconnecting the inter-
;            rupt jumper on the MBB board. Multi-page mode.
;
;  This program writes a formatting code (0e5H) into every
;  byte in the bubble devices. This code is for standard
;  IBM compatible disks.
;
;  The MBB-80 controller base address is read into variable
;  'MB_contbase'. MBB-80 address select pins must correspond
;  to this address. This program uses memory mapped I/O
;  through the base address.
;
;*************************************************************
;*   Jeffrey Neufeld and Michael Hicklin,  CS-03,  Thesis    *
;*************************************************************
;
;* Bdos function numbers for calls *
Bdos_conbuf     equ 10      ;console string input function #
Bdos_conout     equ 2       ;console output char function #
Bdos_pstring    equ 9       ;print string until '$' function #
Bdos_reset      equ 0       ;CP/M-86 reset to CCP function #
;
;* MBB characteristics *
MB_buflen       equ 144     ;buffer length for sector
MB_maxdevs      equ 7       ;bubble devices are #0-#7
MB_maxpages     equ 641     ;# of pages on each bubble device
MB_maxsectors   equ 80      ;# of log sectors on each bub dev
MB_pages_sec    equ 8       ;# of pages per logical sector
MB_pagesize     equ 18      ;bubble device page size
MB_skew         equ 12      ;skew for page translation
;
;* MBB command masks and status masks *
MB_busy_check   equ 00100000B ;cont busy? status check (20H)
MB_init_cmd     equ 00000001B ;init the controller (01H)
MB_int_inhibit  equ 10000000B ;int inhibit/reset mask (80H)
MB_chkint_mask  equ 10000000B ;mask testing if int set (80H)
MB_multi_page   equ 00010000B ;multi-page mode command (10H)
MB_read_cmd     equ 00010010B ;multi-page read command (12H)
MB_reset_cmd    equ 01000000B ;reset the controller (40H)
MB_write_cmd    equ 00010100B ;multi-page write command (14H)
;
;* Miscellaneous equates *
conbuf_size     equ 80      ;size of console input buffer
cr              equ 0dH     ;Ascii carriage return cont char
format_pattern  equ 0e5H    ;format pattern for every byte
lf              equ 0aH     ;Ascii line feed control char
;
```

```
;*****************************************************
;*                MAIN PROGRAM - DRIVER              *
;*****************************************************
;
        CSEG
;
MB80FMT:  call Set_Up           ;do initialization
          cal  Get_Cont_Addr    ;get address of MBB-80 base
          cal  Init_Cont        ;init the cont and devices
  Format_loop:
          call Write_Sector     ;write a sector to bubble
       ;advance to next sector in device,see if last sector
          inc  curr_sector_no   ;increment current sector #
          cmp  curr_sector_no,MB_maxsectors ;last sector ?
          jnz  Format_loop      ;if not, format next sector
       ;was last sector,advance to next bub dev on board
          mov  DX,offset msg_donedev ;addr of done dev msg
          call Print_String     ;write msg to console
          cmp  curr_bub_no,MB_maxdevs ;last bubble on board?
          jz   Done_format      ;if sc, done with formatting
       ;prepare to format next bubble device
          inc  curr_bub_no      ;if not, increment device #
          mov  curr_sector_no,0 ;set sector # back to zero
          jmp  Format_loop      ;go format next device
  Done_format:
          call Close_Up         ;do end of run housekeeping
          mov  CL,Bdos_reset    ;function # for Bdos call
          mov  DL,0             ;parameter to release memory
          call Bdos             ;call Bdos to terminate prog
;
;****************** end of Main Program *******************
;
;
;*****************************************************
;*                BDOS (CP/M-86)  subroutine         *
;*****************************************************
                ;called from: Get_Cont_Addr, Main,
                ;             Print_String, Putchar.
Bdos:           ;** entry to Bdos via software interrupt 224
                ;** parm in - caller loads regs as per req
                ;** parm out - as supplied by Bdos returns
          int   224             ;8086 software interrupt
          ret
;
;*****************************************************
;*                CLOSE_UP  subroutine               *
;*****************************************************
                ;called from: Main.
Close_Up:       ;** issues goodbye
                ;** parm in - none
                ;** parm out - none
       ;issue goodbye message
          call Crlf             ;skip extra line
          mov  DX,offset msg_endformat ;addr done format msg
          call Print_String     ;write msg to console
          ret
;
```

```
;*****************************************************
;*                COMPUTE_PAGENO  subroutine         *
;*****************************************************
                ;called from: Write_Sector.
Compute_Pageno: ;** computes 1st page # for a given sector
                ;** parm in - none, works on curr_sector_no
                ;** parm out - none, updates curr_page_no
          xor   AX,AX             ;set AX to zero
          cmp   AL,curr_sector_no ;is it sector 0 ?
          jz    Store_page        ;if sc, no translation
          xor   CX,CX             ;clear CX for counter
          mov   CL,curr_sector_no ;cntr for translate loop
  Add_skew:
          add   AX,MB_skew        ;# of pages between sectors
          clc                     ;clear carry
          sbb   AX,MB_maxpages    ;mod to # of pages
          jae   Dec_sector        ;jump if positive (CF=0)
          add   AX,MB_maxpages    ;went neg, add back # pages
  Dec_sector:
          loop Add_skew           ;dec sector #, add skew again
  Store_page:
          mov   curr_page_no,AX ;store page number
          ret
;
;*****************************************************
;*                CRLF  subroutine                   *
;*****************************************************
                ;called from: Close_Up, Get_Cont_Addr,
                ;      Init_Cont, Main, Print_String, Set_Up.
Crlf:           ;** sends carriage return,line feed to cons
                ;** parm in - none
                ;** parm out - none
          mov   AL,cr             ;carriage return char
          call  Putchar           ;write it to console
          mov   AL,lf             ;line feed char
          call  Putchar           ;write it to console
          ret
;
;*****************************************************
;*                GET_CONT_ADDR  subroutine          *
;*****************************************************
                ;called from: Main.
Get_Cont_Addr: ;** gets base segment address for the MBB-80
                ;** controller from the user at the console.
                ;** parm in - none
                ;** parm out - none, updates MB_contbase
          mov   DX,offset msg_getaddr ;addr of get cont msg
          call  Print_String      ;write msg to console
       ;get base address keyed in by the user
          mov   CL,Bdos_conbuf    ;input console string func#
          mov   BX,offset cons_buff ;area for cons input
          mov   byte ptr [BX],conbuf_size ;tell Bdos size
          mov   DX,BX             ;load parm for Bdos call
          call  Bdos              ;read from console
          call  Crlf              ;skip a line after input
       ;make sure only four digits keyed in
          mov   BX,offset cons_buff+1 ;byte 1 tells how many
          cmp   byte ptr[BX],4    ;see if exactly four read
          jne   Error_input       ;if not 4, error
       ;make sure all four digits are valid hex
          mov   BX,offset cons_buff+2 ;byte 2 starts data
          xor   AX,AX             ;used for Ascii table index
          mov   CX,4              ;number of digits to check
    Check_valid:
          mov   AL,[BX]           ;move digit to AL for chking
          cmp   AL,030H           ;check to see if too low
```

```
            jb    Error_input
            cmp   AL,046H           ;check to see if too high
            ja    Error_input
            cmp   AL,039H           ;chk mid-invalid (3aH-40H)
            jbe   Valid_hex
            cmp   AL,041H
            jae   Valid_hex
            jmps  Error_input       ;it is in the middle - error
      Valid_hex:
            sub   AX,030H           ;-30H to get table index
            push  BX                ;save buffer addr
            mov   BX,AX             ;AX is index to table
            mov   AL,Ascii_table[BX] ;table look up
            pop   BX                ;restore buffer addr
            mov   byte ptr[BX],AL   ;store hex back in buffer
            inc   BX                ;next digit
            loop  Check_valid       ;go check it
        ;convert 4 valid hex digits to a binary number in AX
            mov   BX,offset conn_buff+2 ;byte 2 starts data
            mov   AH,[BX]           ;get first digit
            mov   CL,4              ;shift it to high nibble
            shl   AH,CL
            inc   BX                ;increment index
            or    AH,[BX]           ;2nd dig or'ed into low nibb
            inc   BX                ;increment index
            mov   AL,[BX]           ;get third digit
            mov   CL,4              ;shift it to high nibble
            shl   AL,CL
            inc   BX                ;increment index
            or    AL,[BX]           ;4th dig or'ed into low nibb
      ;store controller base address that was built in AX
            mov   MB_contbase,AX
            jmps  Get_cont_ret      ;go return
      ;error in input, issue message, retry
      Error_input:
            mov   DX,offset msg_errinp ;addr of error message
            call  Print_String      ;write msg to console
            call  Crlf              ;skip a line
            jmps  Get_Cont_Addr     ;go ask again
      Get_cont_ret:
            ret

;
;*****************************************************************
;*                  INIT_CONT  subroutine                        *
;*****************************************************************
                  ;called from: Main.
Init_Cont:        ;** inits the MBE controller and each device
                  ;** parm in - none
                  ;** parm out - none
        ;initialize page size and mincr loop size
            mov   AX,MB_contbase  ;address of controller base
            mov   ES,AX           ;load ES to address bubble
            mov   AX,MB_maxpages  ;pages per bubble device
            mov   ES:P_loopsize_lo,AL ;loopsize low byte
            mov   ES:P_loopsize_hi,AH ;loopsize hi byte
            mov   ES:P_pagesize_reg,MB_pagesize;page size reg
        ;issue reset command to the controller
            mov   AL,MB_reset_cmd ;reset mask byte
            mov   ES:P_cmnd_reg,AL ;issue reset command
        ;initialize each bubble device
            mov   CX,MB_maxdevs+1 ;count for loop-# of devices
            mov   AL,0            ;device # to initialize
    For_each:
            mov   ES:P_select_bubdev,AL ;select each device
            mov   ES:P_cmnd_reg,MB_init_cmd ;init this device
            push AX!push CX!push ES ;save bubble #,counter,ES
            call  Wait            ;wait for controller to work
```

```
            pop  ES! pop CX! pop AX ;restore ES,cntr,bubble #
            inc  AL                 ;next device number
            loop For_each           ;dec CX, loop if not zero
        ;issue msgs indicating formatting in progress
            call Crlf               ;skip an extra line
            mov  DX,offset msg_formatting ;formatting msg addr
            call Print_String       ;write msg to conscle
            ret
;
;*****************************************************************
;*                  PRINT_STRING  subroutine                     *
;*****************************************************************
                ;called from: Close_Up, Get_Cont_Addr,
                ;             Init_Cont, Main, Set_Up.
Print_String:   ;** prints buffer addressed until '$' hit
                ;** parm in - address of buffer in DX
                ;** parm out - none
            mov  CL,Bdos_pstring ;function # for Bdos call
            call Bdos            ;call Bdos and print
            call Crlf            ;skip a line
            ret
;
;*****************************************************************
;*                  PUTCHAR  subroutine                          *
;*****************************************************************
                ;called from: Crlf.
Putchar:        ;** writes character from AL to console
                ;** parm in - output char in AL
                ;** parm out - none
            mov  CL,Bdos_conout  ;function# for Bdos call
            mov  DL,AL           ;load char to Bdos reg
            call Bdos            ;call Bdos and send
            ret
;
;*****************************************************************
;*                  SET_UP  subroutine                           *
;*****************************************************************
                ;called from: Main.
Set_Up:         ;** inits variables and issues signon msg
                ;** parm in - none
                ;** parm out - none, effects global vars
            call Crlf            ;skip an extra line
            call Crlf            ;skip an extra line
            mov  DX,offset msg_signon ;signon message address
            call Print_String    ;write msg to conscle
            mov  DX,offset msg_version ;version msg address
            call Print_String    ;write msg to console
            call Crlf            ;skip an extra line
        ;initialize all variables and flags
            mov  curr_bub_no,0   ;current bubble # to 0
            mov  curr_sectcr_no,0 ;current sectcr # to 0
            ret
;
;*****************************************************************
;*                  WAIT subroutine                              *
;*****************************************************************
                ;called from: Init_Cont, Write_Sector.
Wait:           ;** checks status of MBB controller for busy
                ;** keeps checking (wait) until not busy
                ;** parm in - none
                ;** parm out - none
            mov  AX,MB_contbase  ;address of controller base
            mov  ES,AX           ;load ES to address bubble
```

```
  See_zero:
          mov   AL,ES:P_status_reg ;get status register
          and   AL,MB_busy_check ;is it all zeros ?
          jz    See_zero         ;if so,keep checking for one
  Cont_busy:
          mov   AL,ES:P_status_reg ;get status register
          and   AL,MB_busy_check ;see if busy, and to mask
          jnz   Cont_busy        ;if busy, check again
          ret
;
;*************************************************************
;*                 WRITE_SECTOR  subroutine                  *
;*************************************************************
                ;called from: Main.
Write_Sector:   ;** writes sector using format patt to MBB80
                ;** parm in - none
                ;** parm out - none
          call Compute_Pageno  ;compute 1st page# of sector
      ;establish addressability to controller
          mov   AX,MB_contbase  ;address of controller base
          mov   ES,AX           ;load ES to address bubble
      ;set multipage mode
          mov   ES:P_cmnd_reg,MB_multi_page ;multipage mode
      ;load first page number for transfer
          mov   AX,curr_page_no ;current page #  formatting
          mov   ES:P_pagesel_lo,AL ;page select lo byte
          mov   ES:P_pagesel_hi,AH ;page select hi byte
      ;set number of pages to transfer = pages/sector
          mov   ES:P_pagecnt_lo,MB_pages_sec ;# pages to xfer
          mov   ES:P_pagecnt_hi,0 ;hi byte of # is zero
      ;set up buffer to send data
          mov   CX,MB_buflen-1  ;count for loop-buffer size
      ;select bubble device and issue write cmd
          mov   AL,curr_bub_no  ;current bubble # formatting
          mov   ES:P_select_bubdev,AL ;select current dev #
          mov   AL,format_pattern ;load format pattern
          mov   ES:P_wdata_reg,AL ;write a byte to FIFO buff
          mov   ES:P_cmnd_reg,MB_write_cmd ;write FIFO buff
      ;wait for interrupt from controller
  Write_int:
          mov AL,ES:P_int_flag    ;get interrupt status
          and AL,MB_chkint_mask   ;has interrupt been set?
          jz Write_Int            ;if not, keep checking
      ;write into MBB FIFO buffer from format_pattern
          mov   AL,format_pattern ;byte from pattern to AL
          mov   ES:P_wdata_reg,AL ;write a byte to FIFO buff
          loop Write_int          ;dec CX, loop if not zero
          push ES                 ;save ES
          call Wait               ;wait for controller to stop
          pop   ES                ;restore ES
          mov   ES:P_cmnd_reg,MB_int_inhibit ;clear cont int
          ret
;
;*************************************************************
;*                   DATA SEGMENT AREA                       *
;*************************************************************
;
      DSEG
                org 0100H     ;leave room for base page
;
;** ------------------Variables--------------------------*
;
Ascii_table     db  00H,01H,02H,03H,04H,05H,06H,07H,08H,09H
                rb  7  ;for Ascii 3aH to 40H - invalid
                db  0aH,0bH,0cH,0dH,0eH,0fH
cons_buff       rb  conbuf_size ;area for console input
curr_bub_no     rb  1           ;bubble device #0-7 formatting
```

```
curr_page_no    rw  1          ;bubble page # formatting
curr_sector_no  rb  1          ;bub logic sect # formatting
MB_contbase     dw  0000H      ;base segment addr for MBB-80
;
;**-------- string data area for console messages ---------*
;
msg_donedev     db  '  Done with a device.$'
msg_endformat   db  '*Formatting complete...'
                db  'returning to CP/M!$'
msg_errinp      db  '**ERROR: not exactly 4 digits entered,'
                db  ' or invalid hex digits!!$'
msg_formatting  db  'Formatting the devices.......$'
msg_getaddr     db  cr,lf,'Key in 4 digit segment base addr'
                db  'ess for MBB-80 controller.',cr,lf
                db  'Must be in hex (4 digits, then CR only)'
                db  ' => $'
msg_signon      db  '                '
                db  '** MBE-80 CP/M-86 BUBBLE FORMATTER **$'
msg_version     db  '                '
                db  'Multi-Page Mode   Version 1.0$'
                db  0          ;GENCMD to fill last address
;
;**************** end of variables ***********************
;
        ESEG
;
;*****************************************************************
;*                  MBB-80 CONTROLLER AND PORTS                  *
;*****************************************************************
;
P_pagesel_lo    rb  1          ;ls byte for page select, (0)
P_pagesel_hi    rb  1          ;ms 2 bits for page select,(1)
P_cmnd_reg      rb  1          ;command register, (2)
P_rdata_reg     rb  1          ;read data register, (3)
P_wdata_reg     rb  1          ;write data register, (4)
P_status_reg    rb  1          ;status register, (5)
P_pagecnt_lo    rb  1          ;ls byte for page counter, (6)
P_pagecnt_hi    rb  1          ;ms 2 bits for page cnter, (7)
P_lcopsize_lo   rb  1          ;ls byte for minor lcop sz,(8)
P_loopsize_hi   rb  1          ;ms 2 bits for min loop sz,(9)
                rw  1          ;internal use(page pcs), (A,B)
P_pagesize_reg  rb  1          ;page size register, (C)
                rw  1          ;TI use only, (D,E)
P_select_bubdev rb  1          ;two uses: sel bubble dev (F)
P_int_flag      equ P_select_bubdev ;    interrupt flag (F)
;********** end of Controller and Port definitions *********
;
;*****************************************************************
;*                  End of Program MB80FMT                       *
;*****************************************************************
                END
```

APPENDIX F

PROGRAM LISTING OF MBBIOS.A86

```
;
;FILENAMES:  Pascal = MB.BIOS.TEXT
;            CP/M = MBBIOS.A86
;
;
        title   'Customized Basic I/O System'
;*****************************************************************
;*                                                               *
;* This Customized BIOS adapts CP/M-86 to                        *
;* the following hardware configuration:                         *
;*     Processor:  iSBC 86/12A                                   *
;*     Disk Controller:  Intel SBC 202                           *
;*     Bubble memory: MBB-80 with memory-mapped I/O              *
;*     Memory model:  8080                                       *
;*                                                               *
;*     Programmers: J.A. Neufeld, M.S. Hicklin                   *
;*     Revisions :                                               *
;*                                                               *
;*****************************************************************


;********************   EQUATES   ***************************
;*                                                             *

;-------------------- Miscellaneous equates ------------------
;|                                                            |
   addr_high_ram    equ 0f00H ;high para user available RAM
   bdos_int_type    equ 224   ;reserved BDOS interrupt
   cr               equ 0dH   ;Ascii carriage return
   disk_type        equ 01H   ;type for standard floppy disk
   true             equ -1    ;for conditional assembly
   false            equ not true ;for conditional assembly
   lf               equ 0aH   ;Ascii line feed
   max_retries      equ 10    ;for disk I/O, # of tries
   mbb80_type       equ 02H   ;type for MBB-80 bubble
   sector_size      equ 128   ;CP/M logical disk sector size
;|                                                            |
;-------------------------------------------------------------


;-------------- I8251 USART console ports --------------------
;|                                                            |
   CONP_data        equ 0d8H  ;I8251 data port
   CONP_status      equ 0daH  ;I8251 status port
;|                                                            |
;-------------------------------------------------------------


;--- Disk Controller command bytes and masks (iSBC 202) ---
;|                                                            |
   DK_chkint_mask   equ 004H  ;mask to check for DK interupt
   DK_home_cmd      equ 003H  ;move to home position command
   DK_read_cmd      equ 004H  ;read command
   DK_write_cmd     equ 006H  ;write command
;|                                                            |
;-------------------------------------------------------------
```

```
;--------- INTEL iSBC 202 Disk Controller Ports ----------
;|                                                        |
   DKP_base         equ 078H        ;ctrler's base in CP/M-86
   DKP_result_type  equ DKP_base+1  ;operation result type
   DKP_result_byte  equ DKP_base+3  ;operation result byte
   DKP_reset        equ DKP_base+7  ;disk reset
   DKP_status       equ DKP_base    ;disk status
   DKP_iopb_low     equ DKP_base+1  ;low addr byte of iopb
   DKP_iopb_high    equ DKP_base+2  ;high addr byte of iopb
;|                                                        |
;--------------------------------------------------------


;-------- Magnetic bubble characteristics (MBB-80) --------
;|                                                        |
   MB_buflen        equ 144    ;buffer length for MBB sector
   MB_maxdevs       equ 7      ;bubble devices are #0-#7
   MB_maxpages      equ 641    ;# of pages on each device
   MB_maxsectors    equ 80     ;# of log. sectors on each dev
   MB_pages_sec     equ 8      ;# of pages per logical sector
   MB_pagesize      equ 18     ;bubble device page size
   MB_skew          equ 12     ;skew factor for page xlation
;|                                                        |
;--------------------------------------------------------


;---- Magnetic bubble command bytes and masks (MBB-80) ----
;|                                                        |
   MB_chkbusy_cmd   equ 020H   ;is controller busy ? status
   MB_chkint_mask   equ 080H   ;mask to chk for MBB interupt
   MB_inhint_cmd    equ 080H   ;interrupt inhibit/reset mask
   MB_init_cmd      equ 01H    ;initialize the controller
   MB_mpage_cmd     equ 010H   ;multi-page mode operation cmd
   MB_read_cmd      equ 012H   ;multi-page read command
   MB_reset_cmd     equ 040H   ;reset the controller
   MB_write_cmd     equ 014H   ;multi-page write command
;|                                                        |
;--------------------------------------------------------


;----------------- Starting addresses --------------------
;|                                                        |
;  Loader_bios is true if assembling the
;  LCADER_BIOS, otherwise BIOS is for the
;  CPM.SYS file. This section will assign the
;  appropriate equates to the starting addresses.
;
   loader_bios      equ false ;** controls conditional asm
;
        IF      not loader_bios
   addr_bdos        equ 0B06H ;BDOS entry point in CCP
   addr_bios        equ 2500H ;start of BIOS after CCP
   addr_ccp         equ 0000H ;base of CCP is 0
        ENDIF   ;not loader_bios
;
        IF      loader_bios
   addr_bdos        equ 0406H ;stripped BDOS entry in CCP
   addr_bios        equ 1200H ;start of LDBIOS after CCP
   addr_ccp         equ 0003H ;base of CPMLOADER
        ENDIF   ;loader_bios
;|                                                        |
;--------------------------------------------------------


;*                                                       *
;******************* End of Equates ***********************
```

```
;****************** START CP CODE ************************
        CSEG
        orq     addr_ccp
CCP:
        orq     addr_bios

;-------BIOS Jump Vector for Individual Routines ----------
;|                                                         |
        jmp INIT        ;enter from BOOT ROM or LOADER
        jmp WBOOT       ;arrive here from BDOS call 0
        jmp CONST       ;return console keyboard status
        jmp CONIN       ;return console keyboard char
        jmp CONOUT      ;write char to console device
        jmp LISTOUT     ;write character to list device
        jmp PUNCH       ;write character to punch device
        jmp READER      ;return char from reader device
        jmp HOME        ;move to trk 00 on cur sel drive
        jmp SELDSK      ;select disk for next rd/write
        jmp SETTRK      ;set track for next rd/write
        jmp SETSEC      ;set sector for next rd/write
        jmp SETDMA      ;set offset for user buff (DMA)
        jmp READ        ;read a 128 byte sector
        jmp WRITE       ;write a 128 byte sector
        jmp LISTST      ;return list status
        jmp SECTRAN     ;xlate logical->physical sector
        jmp SETDMAB     ;set segment base for buff (DMA)
        jmp GETSEGT     ;return offset of Mem Desc Table
        jmp GETIOBF     ;return I/O map byte (iobyte)
        jmp SETIOBF     ;set I/O map byte (iobyte)
;|                                                         |
;---------------------------------------------------------



;*********************************************************
;**********   'INIT' jump vector destination   ***********
;*********************************************************
                ;called from: bios jump vector.
INIT:           ;** Enter from BOOT ROM or LOADER
                ;** parm in - none
                ;** parm out - none
    ;print signon message and initialize hardware
        mov AX,CS       ;we entered with a JMPF so use
        mov SS,AX       ;CS: as the initial value of SS:,
        mov DS,AX       ;DS:,
        mov ES,AX       ;and ES:
    ;use local stack during initialization
        mov SP,offset stack_base
        cld             ;auto-increment on
    ;setup all interrupt vectors in low memory to
    ;address the soft/hardware traps.
    ;
            IF      not loader_bios
        call Init_Bios_Int ;set up interrupts for CPM.SYS
            ENDIF       ;not loader_bios
    ;
            IF      loader_bios
        call Init_Ldr_Int  ;set up interrupts for LOADER
            ENDIF       ;loader_bios
    ;
    ;perform special initializations for CP/M-86
        call Load_Dma_Addr ;load dma addr for devices
        call Device_Inits  ;init all devices
```

```
     ;(calls for additional initialization go here)
        mov BX,offset msg_signon
        call Print_Msg   ;print signon message
        mov CL,0         ;default to dr A: on coldstart
        jmp CCP          ;jump to cold start entry of CCP
;


;*************************************************************
;***********  'WBOOT'  jump vector destination  **************
;*************************************************************
                ;called from: bios jump vector.
WBCCT:          ;** Arrive here from BDOS call number 0
                ;** parm in - none
                ;** parm out - none
        jmp CCP+6        ;entry to CCP at command level
;


;*************************************************************
;*                                                           *
;*         CP/M Character I/O Interface Routines             *
;*   Console is USART (I8251A) on 8612 at ports D8/DA        *
;*                                                           *
;*************************************************************

;*************************************************************
;***********  'CONST' jump vector destination  ***************
;*************************************************************
                ;called from: bios jump vector.
CONST:          ;** returns console keyboard status
                ;** parm in - none
                ;** parm out - returns status in AL
                ;**              00=not ready, 0ff=ready
        in AL,CONP_status ;get status
        and AL,2          ;see if ready-bit 1-is set
        jz Const_ret      ;if not, it is zero and not ready
        or AL,0ffH        ;is ready, return non-zero
   Const_ret:
        ret
;


;*************************************************************
;***********  'CONIN' jump vector destination  ***************
;*************************************************************
                ;called from: bios jump vector.
CONIN:          ;** returns console keyboard character
                ;** parm in - none
                ;** parm out - returns character in AL
        call CONST        ;get console status
        test AL,AL        ;is it zero (not ready)?
        jz CONIN          ;if zero, keep checking
        in AL,CONP_data   ;ready, so read character
        and AL,07fH       ;remove parity bit
        ret
;
```

```
;*************************************************************
;***********  'CONOUT' jump vector destination  ***********
;*************************************************************
                    ;called from: bios jump vector.
CONOUT:             ;** write character to console keyboard.
                    ;** parm in - character to be output in CL
                    ;** parm out - none
        in AL,CONP_status ;get console status
        and AL,1          ;see if ready-bit 0-is set
        jz CONOUT         ;if zero, not ready-keep checking
        mov AL,CL         ;load input parm to AL for out
        out CONP_data,AL  ;output character to console
        ret
;



;*************************************************************
;***********  'LISTOUT' jump vector destination  ***********
;*************************************************************
                    ;called from: bios jump vector.
LISTOUT:            ;** write character to list device.
                    ;** parm in - none
                    ;** parm out - char to be output in CL
     ;not implemented
        ret
;



;*************************************************************
;***********  'LISTST' jump vector destination  ***********
;*************************************************************
                    ;called from: bios jump vector.
LISTST:             ;** returns the list status.
                    ;** parm in - none
                    ;** parm out - list device status in AL
                    ;                00=not ready, 0ff=ready
     ;not implemented
        ret
;



;*************************************************************
;***********  'PUNCH' jump vector destination  ***********
;*************************************************************
                    ;called from: bios jump vector.
PUNCH:              ;** write character to the punch device.
                    ;** parm in - character to send in CL
                    ;** parm out - none
     ;not implemented
        mov AL,01aH     ;return eof for now
        ret
;
```

```
;*************************************************************
;**********  'READER' jump vector destination  **********
;*************************************************************
                ;called from: bios jump vector.
READER:         ;** return character from reader device.
                ;** parm in - none
                ;** parm out - character read in AL
        mov AL,01aH     ;return eof for now
        ret
;



;*************************************************************
;********** 'GETIOBF' jump vector destination  **********
;*************************************************************
                ;called from: bios jump vector.
GETIOBF:        ;** return I/O map byte (iobyte)
                ;** parm in - none
                ;** parm out - returns iobyte in AL
        mov AL,iobyte   ;iobyte not implemented
        ret
;



;*************************************************************
;********** 'SETIOBF' jump vector destination  **********
;*************************************************************
                ;called from: bios jump vector.
SETIOBF:        ;** set I/O map byte (iobyte)
                ;** parm in - iobyte to be set in CL
                ;** parm out - none
        mov iobyte,CL   ;iobyte not implemented
        ret
;



;*************************************************************
;*                                                           *
;*                  Disk Input/Output Routines               *
;* Disk is i202 Controller with ports at 078H for 8 bytes    *
;*                                                           *
;*************************************************************

;*************************************************************
;**********  'SELDSK' jump vector destination  **********
;*************************************************************
                ;called from: bios jump vector.
SELDSK:         ;** select disk for next read/write
                ;** parm in - disk number to select in CL
                ;** parm out - address of first dph in BX
                ;   dph is a disk parameter header.
        mov disk,CL     ;save disk number
        mov BX,0        ;ready for error return
        cmp CL,num_log_disks ;beyond max disks?
        jnb Seldsk_ret  ;return if so
        mov CH,0        ;double(n)
        mov BX,CX       ;BX = n
        mov CL,4        ;ready for *16, 16 bytes each dph
```

```
        shl BX,CL           ;n = n * 16
        mov CX,offset dpbase ;address of first dph
        add BX,CX           ;dpbase + n * 16
        push BX             ;save dpbase
    ;determine type of device this disk number is
        xor BX,BX           ;clear BX of index
        mov BL,disk         ;load disk number for index
        mov AL,device_table[BX] ;find type of device
        mov device_type,AL  ;store the type returned
    ;make CP/M logical disk # mapping to floppy cont or
    ;MBB-80 cont address depending on device type.
        cmp device_type,disk_type ;is this a floppy?
        jne Load_mbb80_cont ;if not, do MBB-80 cont addr
        mov AL,DK_logical_table[BX] ;get floppy disk #
        mov DK_disk,AL      ;store floppy cont disk #
        jmps Seldsk_ret     ;go return
Load_mbb80_cont:
        add BL,BL           ;double disk # for word index
        mov AX,MB_logical_table[BX] ;get addr of cont
        mov MB_contbase,AX  ;store as current base addr
Seldsk_ret:
        pop BX              ;restore dpbase for return
        ret
;



;*****************************************************
;**********   'HOME' jump vector destination   **********
;*****************************************************
                ;called from: bios jump vector.
HOME:           ;** move to trk 0 on curr selected drive
                ;** parm in - none
                ;** parm out - none
        cmp device_type,disk_type ;is this a floppy disk?
        jne Mbb80_home      ;if not, home bubble
        mov DK_io_com,DK_home_cmd ;home the floppy disk
        mov track,0
        call Dk_Execute_Cmd
        jmps Home_ret       ;go return
Mbb80_home:
        xor CX,CX           ;clear CX, parm - track=0
        call SETTRK         ;set track for bubble = 0
Hcme_ret:
        ret
;



;*****************************************************
;**********  'SETTRK' jump vector destination  **********
;*****************************************************
                ;called from: bios jump vector, HOME.
SETTRK:         ;** Set track for next read/write
                ;** parm in - track address in CX (CL)
                ;** parm out - none
        mov track,CL        ;store track number
        cmp device_type,disk_type ;is this a floppy disk?
        je Settrk_ret       ;if so, just return
        call Mbb80_Track_Xlat ;bubble, so xlat track->bub#
Settrk_ret:
        ret
;
```

```
;*************************************************************
;**********  'SETSEC' jump vector destination  **********
;*************************************************************
                ;called from: bios jump vector.
SETSEC:         ;** Set sector for next read/write
                ;** parm in - sector number in CX (CL)
                ;** parm out - none
        mov sector,CL   ;stcre sector number
        ret
;



;*************************************************************
;********** 'SECTRAN' jump vector destination  **********
;*************************************************************
                ;called from: bios jump vector.
SECTRAN:        ;** Translate logical to physical sector
                ;** parm in - sector in CX; table at [DX]
                ;** parm out - physical sector # in BX
        mov CH,0        ;clear high byte
        mov BX,CX       ;load input parm for return
        test DX,DX      ;is there a xlat to be done ?
        jz No_skew      ;if not, just return
        add BX,DX       ;add sector to tran table address
        mov BL,[BX]     ;get logical sector
        jmps Sectran_ret ;go return
   No_skew:
        add BX,1        ;no xlat,CP/M sect #0 => sect #1
   Sectran_ret:
        ret
;



;*************************************************************
;**********  'SETDMA' jump vector destination  **********
;*************************************************************
                ;called from: bios jump vector.
SETDMA:         ;** Set offset for user DMA buffer
                ;** parm in - DMA offset in CX
                ;** parm out - none
        mov dma_offset,CX  ;store dma offset
        call Load_Dma_Addr ;update DMA info for all devices
        ret
;



;*************************************************************
;********** 'SETDMAB' jump vector destination  **********
;*************************************************************
                ;called from: bios jump vector.
SETDMAB:        ;** Set segment base for DMA buffer
                ;** parm in - segment in CX
                ;** parm out - none
        mov dma_segment,CX ;store dma segment
        call Load_Dma_Addr ;update DMA info for all devices
        ret
;
```

```
;*****************************************************
;**********  'GETSEGT' jump vector destination **********
;*****************************************************
                ;called from: bios jump vector.
GETSEGT:        ;** Return offset of memory desc table
                ;** parm in - none
                ;** parm out - address of table in BX
        mov BX,offset mem_desc_table
        ret
;



;*****************************************************
;*                                                   *
;*  All I/O parameters are setup:                    *
;*     disk      is disk number       (SELDSK)       *
;*     track     is track number      (SETTRK)       *
;*     sector    is sector number     (SETSEC)       *
;*  Each device maintains its own DMA info as required *
;*  by its controller, using dma_offset and dma_segment.*
;*  READ reads the selected sector to the DMA address, *
;*  and WRITE writes the data from the DMA address to  *
;*  the selected sector. The MBB-80 bubble will use diff-*
;*  erent routines to perform the read and write funct-*
;*  ions. The MBB-80 works with MB_bub_no (from MBB_Track_*
;*  Xlat) and MB_page_no (from Mbb_Sector_Xlat) - these *
;*  values are derived from the vars, track and sector.*
;*                                                   *
;*****************************************************
;
;*****************************************************
;**********   'READ' jump vector destination   **********
;*****************************************************
                ;called from: bios jump vector.
READ:           ;** Read a 128 byte sector
                ;** parm in - none
                ;** parm out - return code in AL
                ;   00 = OK, FF = unsuccessful
        cmp device_type,disk_type ;is this a floppy disk?
        jne Bubble_read     ;If not, use bubble routine
        mov CL,4
        mov AL,DK_disk      ;combine disk selection
        sal AL,CL           ;with opcode
        or AL,DK_read_cmd   ;create iopb for read
        mov DK_io_com,AL    ;load iopb
        call DK_Execute_Cmd ;perform the read
        jmps Read_ret       ;return
   Bubble_read:             ;use bubble routine to read
        call Mbb80_Read     ;perform the read
   Read_ret:
        ret
;
```

```
;*************************************************************
;**********  'WRITE' jump vector destination    **********
;*************************************************************
                    ;called from: bios jump vector.
WRITE:              ;** Writes a 128 byte sector
                    ;** parm in - none
                    ;** parm out - return code in AL
                    ;   00 = OK, FF = unsuccessful
        cmp device_type,disk_type ;is this a floppy disk?
        jne Bubble_write    ;If not, use bubble routine
        mov CL,4
        mov AL,DK_disk      ;combine disk selection
        sal AL,CL           ;with opcode
        or AL,DK_write_cmd  ;create iopb for write
        mov DK_io_com,AL    ;load iopb
        call DK_Execute_Cmd ;perform the write
        jmps Write_ret      ;return
   Bubble_write:            ;use bubble routine to write
        call Mbb80_Write    ;perform the write
   Write_ret:
        ret
;



;*************************************************************
;*                                                           *
;* The following subroutines perform various specific        *
;* tasks for the above jump vectors.                         *
;*                                                           *
;*************************************************************

;*************************************************************
;*                DEVICE_INITS  subroutine                   *
;*************************************************************
                    ;called from: INIT,
Device_Inits:       ;** Perform any init necssary for
                    ;   all devices generated.
                    ;** parm in - none
                    ;** parm out - none
     ;(*** Device initialization for the iSBC 202 disk ***)
     ;load address of the iSBC 202 iopb (channel command)
        mov CL,4            ;load CL for shift
        mov AX,CS           ;load AX with this segment
        sal AX,CL           ;move segment to high byte
        add AX, offset DK_iopb ;offset of iopb (chan cmd)
        mov DK_iopb_addr,AX ;store for later use
     ;see if any iSBC 202 controller to be initialized
        xor CX,CX           ;clear CX for counter in loop
        mov CL,num_log_disks ;load # of disk devices
   Check_i202:
        mov BX,CX           ;index into device table
        cmp device_table[BX],disk_type ;i202 disk?
        je  Init_i202       ;if so, go init the contr'ler
        loop Check_i202 ;check next
        jmps Done_i202  ;no i202, go init mbb80s
   Init_i202:
        in AL,DKP_result_type ;clear the controller
        in AL,DKP_result_byte
        out DKP_reset,AL ;AL is dummy for this command
     ;(*** Device initialization for the MBB-80 bubble ***)
     ;initialize each MBB-80 controller defined
   Done_i202:
        xor CX,CX           ;clear CX for counter in loop
        mov CL,num_log_disks ;load # of disk devices
        push ES             ;save register
   Init_mbb80:
```

```
        xor BX,BX           ;clear BX for index
        mov BL,CL           ;load cont # to BX
        dec BX              ;subtract 1 for table
        add BL,BL           ;double index for word lookup
        mov AX,MB_logical_table[BX] ;get cont addr
        cmp AX,MB_null      ;Is it a null addr (place holder)?
        je  Next_mbb80      ;if so, go to next cont'ler
        mov MB_contbase,AX  ;load to current base
    ;initialize page size and minor loop size
        mov ES,AX                   ;load ES to address bubble
        mov AX,MB_maxpages          ;pages per bubble device
        mov ES:MBP_loopsize_lo,AL   ;loopsize low byte
        mov ES:MBP_loopsize_hi,AH   ;loopsize hi byte
        mov ES:MBP_pgsize_reg,MB_pagesize ;load page size
    ;issue reset command to the controller
        mov AL,MB_reset_cmd         ;reset mask byte
        mov ES:MBP_cmnd_reg,AL      ;issue reset command
    ;initialize each bubble device
        push CX                     ;save CX, outer counter
        mov CX,MB_maxdevs+1         ;count for loop-# of devs
        mov AL,0                    ;device # to initialize
    For_each:
        mov ES:MBP_select_bub,AL    ;select each device
        mov ES:MBP_cmnd_reg,MB_init_cmd ;init this device
        push AX!push CX!push ES     ;save bubble#,counter,ES
        call Mbb80_Wait             ;wait for controller
        pop ES! pop CX! pop AX      ;restore ES,counter,bubble#
        inc AL                      ;next device number
        loop For_each               ;dec CX, loop if not zero
        pop CX                      ;restore CX, outer counter
    Next_mbb80:
        loop Init_mbb80             ;go init next cont
        pop ES                      ;restore register
    Device_ret:
        ret
;
;
;*****************************************************
;*            DK_EXECUTE_CMD  subroutine             *
;*****************************************************
                    ;called from: READ, WRITE.
Dk_Execute_Cmd:     ;** Executes a disk read/write command
                    ;** parm in - none
                    ;** parm out - status of the op in AL.
                    ;**  00= OK, FF= unsuccessful
    Load_retries:
        mov DK_rtry_cnt,max_retries ;load count for retries
      ;send iopb to disk controller via two ports (2 bytes)
    Send_iopb:
        in AL,DKP_result_type ;clear the controller
        in AL,DKP_result_byte ;clear the controller
        mov AX,DK_iopb_addr ;get address of iopb
        out DKP_iopb_low,AL ;output low byte of iopb addr
        mov AL,AH           ;load high byte to AL for output
        out DKP_iopb_high,AL ;out high byte of iopb addr
      ;check for interrupt from disk controller
    Disk_int:
        in AL,DKP_status ;get disk status
        and AL,DK_chkint_mask ;interrupt set?
        jz Disk_int     ;if not, keep checking
      ;see if interrupt signifies I/O completion
        in AL,DKP_result_type ;get reason for interrupt
        cmp AL,00H      ;was I/O complete ?
        jz Check_result ;if so, go check the result byte
        in AL,DKP_result_byte ;clear result byte
        mov AL,080H     ;disk wasn't ready - load code
        jmps Retry      ;load err code, and go retry
      ;check result byte for errors
```

```
    Check_result:
         in AL,DKP_result_byte ;get result byte
         and AL,0feH      ;check for error in any bit
         jnz Retry        ;found one, retry
      ;read or write is ok, AL contains 0 for return
         jmps Dk_execute_ret
      ;retry the command until max_retries attempted.
    Retry:
         mov DK_err_code,AL ;save error result byte
         dec DK_rtry_cnt ;dec number of attempts so far
         jnz Send_iopb    ;if not zero, send command again
      ;did max_retries, no success - issue error message
         call Dk_Print_Err ;print out appropriate err msg
         in AL,CONP_data ;flush usart receiver buffer
         call Ucon_Echo  ;read upper case console character
         cmp AL,'C'
         je Wboot_jump   ;cancel
         cmp AL,'R'
         je Load_retries ;retry max times again
         cmp AL,'I'
         je Dk_execute_ret ;ignore error
         or AL,0FFH      ;set code for permanent error
         jmps Dk_execute_ret
    Wboot_jump:          ;can't make it w/ a short jump
         jmp WBOOT
    Dk_execute_ret:
         ret
;
;*****************************************************
;*              DK_PRINT_ERR  subroutine             *
;*****************************************************
                  ;called from: Dk_Execute_Cmd.
Dk_Print_Err:     ;** Prints out disk error messages.
                  ;** parm in - uses DK_err_code
                  ;** parm out - none
         mov BL,DK_err_code ;load code for index to table
         mov BH,0         ;clear high byte of index
         test BL,0fH      ;see if error bits in low nibble
         jz Use_hi_index ;error is in high nibble
    Use_low_index:
         mov BL,DK_err_loinx[BX] ;get offset in addr table
         jmps Print_it    ;go print the message
    Use_hi_index:
         mov CL,4         ;shift four bits right
         shr BX,CL        ;shift it right
         mov BL,DK_err_hiinx[BX] ;get offset in addr table
    Print_it:
         mov BX,DK_err_table[BX] ;load addr of message
         call Print_Msg   ;print appropriate message
         ret
;
;
         IF       not loader_bios
;*****************************************************
;*              INIT_BIOS_INT  subroutine            *
;*****************************************************
                  ;called from: INIT. (if not loader_bios)
Init_Bios_Int:    ;** sets up the interrupt vectors in low
                  ;**  memory to vector soft/hard interrupts.
                  ;** parm in - none
                  ;** parm out - none
         push DS!push ES ;save the DS & ES register
         mov iobyte,0     ;clear iobyte
         mov AX,0
         mov DS,AX
         mov ES,AX        ;set ES and DS to zero
```

```
      ;setup interrupt 0 to address trap routine
         mov int0_offset,offset Trap_Handler
         mov int0_segment,CS
         mov DI,4
         mov SI,0          ;then propagate
         mov CX,510        ;trap vector to
         rep movs AX,AX    ;all 256 interrupts
      ;BDOS offset to proper interrupt
         mov bdos_int_offset,addr_bdos
         pop ES ! pop DS ;restore the ES & DS register
         ret
         ENDIF   ;not loader_bios
;
;
         IF      loader_bios
;*****************************************************
;*                INIT_LDR_INT  subroutine           *
;*****************************************************
                ;called from: INIT. (if loader_bios)
Init_Ldr_Int:   ;** sets up the interrupt vectors in low
                ;**  memory to vector soft/hard interrupts.
                ;** parm in - none
                ;** parm out - none
      ;BDOS offset to proper interrupt
         push DS           ;save the DS register
         mov AX,0          ;set to absolute low memory
         mov DS,AX         ;make it addressable
         mov bdos_int_offset,addr_bdos ;offset
         mov bdos_int_segment,CS       ;this segment
         pop DS            ;restore DS register
      ;issue message telling where loading from
         mov BX, offset msg_i202 ;assume i202
         cmp device_table,disk_type ;check default disk
         je  Print_loader ;is disk, print msg
         mov BX, offset msg_mbb ;its the mbb80
   Print_loader:
         call Print_msg  ;write msg to console
      ;(additional Loader initializations go here )
         ret
         ENDIF   ;if loader_bios
;
;*****************************************************
;*                LOAD_DMA_ADDR  subroutine          *
;*****************************************************
                ;called from: INIT, SETDMA, SETDMAB.
Load_Dma_Addr:  ;** upon new DMA addr, updates all device's
                ;**  DMA words, channel commands, etc., that
                ;**  are needed because of a new DMA addr.
                ;** parm in - none, operates using variables
                ;**    dma_offset and dma_segment.
                ;** parm out - none, updates var DK_dma_addr
      ;update iSBC 202 disk controller dma address
         mov CL,4          ;iSBC 202 uses 16-bit address
         mov AX,dma_segment ;load segment
         sal AX,CL         ;move segment to high bits
         add AX,dma_offset ;add in dma offset
         mov DK_dma_addr,AX ;store new dma addr - disk
      ;MBB-80 uses 20-bit address, therefore can use the
      ;dma_segment and dma_offset variables directly.
         ret
;
;
```

```
;*****************************************************
;*                  MBB80_READ  subroutine           *
;*****************************************************
                    ;called from: READ.
Mbb80_Read:         ;** reads a sector from bubble
                    ;** parm in - none
                    ;** parm out - status of the op in AL.
                    ;**  00= OK, FF= unsuccessful
        push ES                     ;save register
        call Mbb80_Sector_Xlat      ;compute 1st page# of sect
    ;establish addressability to controller
        mov AX,MB_contbase          ;address of controller base
        mov ES,AX                   ;load ES to address bubble
    ;set multipage mode
        mov ES:MBP_cmnd_reg,MB_mpage_cmd  ;multipage mode cmd
    ;load first page number for transfer
        mov AX,MB_page_no           ;current page number
        mov ES:MBP_pagesel_lo,AL ;page select lo byte
        mov ES:MBP_pagesel_hi,AH ;page select hi byte
    ;set number of pages to transfer = pages/sector
        mov ES:MBP_pagecnt_lo,MB_pages_sec ;#pages to xfer
        mov ES:MBP_pagecnt_hi,0            ;hi byte of # is 0
    ;set up dma address to receive data
        mov CX,MB_buflen            ;count for loop-buffer size
        push DS                     ;save CP/M's DS
        mov AX,dma_segment          ;get dma segment
        push AX                     ;save dma segment DS
        mov BX,dma_offset           ;offset of dma area
    ;select bubble device and issue read command
        mov AL,MB_bub_no            ;current bubble number
        pop DS                      ;done local, readdr dma area
        mov ES:MBP_select_bub,AL;select current dev #
        mov ES:MBP_cmnd_reg,MB_read_cmd ;issue read from FIFO
    ;wait for interrupt from controller
Read_int:
        mov AL,ES:MBP_int_flag   ;get interrupt status
        and AL,MB_chkint_mask    ;interrupt set ?
        jz Read_int              ;if zero, keep checking
    ;see if read enough from bubble sector to fill dma area
        cmp CX,(MB_buflen - sector_size) ;transferred enough?
        jnz Read_one             ;if not, read another byte
        pop DS                   ;restore CP/M's DS
        mov BX,offset MB_overflow ;reset dest to overflow
    ;read from MBB FIFO buffer into dma area
Read_one:
        mov AL,ES:MBP_rdata_reg ;read a byte into accum
        mov [BX],AL              ;load accum into dma area
        inc BX                   ;increment index
        loop Read_int            ;dec CX, loop if not zero
        push ES                  ;save ES for call
        call Mbb80_Wait          ;wait for controller
        pop ES                   ;restore ES after call
        mov ES:MBP_cmnd_reg,MB_inhint_cmd  ;clear cont int
        pop ES                   ;restore register
        mov AL,0                 ;indicate success
        ret

;
;
```

```
;*****************************************************
;*            MBB80_SECTOR_XLAT  subroutine          *
;*****************************************************
                    ;called from: Mbb80_Read, Mbb80_Write.
Mbb80_Sector_Xlat:  ;** computes 1st page# for a given sector
                    ;** on a single chip. Based on 80 sectors
                    ;** on each chip - sector = 128 bytes.
                    ;** parm in - none, works on sector
                    ;** parm out - none, updates MB_page_no
        xor AX,AX                   ;set AX to 0 to hold page#
        xor CX,CX                   ;clear CX for counter
        mov CL,sector               ;ctr for translation loop
        xor DX,DX                   ;clear DX
        mov DL,MB_sector            ;sect# for 1st sect on trk
        add CX,DX                   ;add 1st sect# to log sect#
        dec CL                      ;subtract 1 for the loop
        jz Mbb80_sx_exit            ;sect 1 is page 0, no xlat
   Add_skew:
        add AX,MB_skew              ;add skew between pages
        clc                         ;clear carry
        sbb AX,MB_maxpages          ;mod to # of pages
        jae Dec_sector              ;jump if positive (CF=0)
        add AX,MB_maxpages          ;went (-), add back #pages
   Dec_sector:
        loop Add_skew               ;dec sector#,add skew again
   Mbb80_sx_exit:
        mov MB_page_no,AX           ;store page number
        ret
;
;*****************************************************
;*            MBB80_TRACK_XLAT  subroutine           *
;*****************************************************
                    ;called from: SETTRK.
Mbb80_Track_Xlat:   ;** computes bubble # from track #. Gets
                    ;** first bubble sector (1-80) for that
                    ;** track for later conversion to page #.
                    ;** parm in - none, works on track.
                    ;** prm out - loads MB_bub_no,MB_sector
        xor BX,BX                   ;clear BX for add
        mov BL,track                ;load track - index
        add BL,BL                   ;double track# for index
        mov AX,MB_track_table[BX]   ;get word from table
        mov MB_bub_no,AH            ;low byte = bubb device#
        mov MB_sector,AL            ;high byte = 1st sector#
        ret
;
;*****************************************************
;*              MBB80_WAIT  subroutine               *
;*****************************************************
                    ;called from: Mbb80_Init, Mbb80_Read,
                    ;             Mbb80_Write.
Mbb80_Wait:         ;** checks status of MBB cont for busy
                    ;** keeps checking (wait) until not busy
                    ;** parm in - none
                    ;** parm out - none
        mov AX,MB_contbase          ;address of cont base
        mov ES,AX                   ;load ES to addr bubble
   See_zero:
        mov AL,ES:MBP_status_reg    ;get status register
        and AL,MB_chkBusy_cmd       ;is it all zeros ?
        jz See_zero                 ;if so, keep checking
   Cont_busy:
        mov AL,ES:MBP_status_reg    ;get status register
        and AL,MB_chkBusy_cmd       ;see if busy, and to mask
        jnz Cont_busy               ;if busy, check again
        ret
;
```

```
;*****************************************************
;*                  MBB80_WRITE  subroutine          *
;*****************************************************
                    ;called frcm: WRITE.
Mbb80_Write:        ;** writes a sector to bubble
                    ;** parm in - none
                    ;** parm out - status of the op in AL.
                    ;**   00= CK, FF= unsuccessful
     IF not loader_bios
        push ES                     ;save register
        call Mbb80_Sector_Xlat      ;get 1st page# of sector
     ;establish addressability to controller
        mov AX,MB_contbase          ;address of controller base
        mov ES,AX                   ;load ES to address bubble
     ;set multipage mode
        mov ES:MBP_cmnd_reg,MB_mpage_cmd  ;multipg mode cmd
     ;load first page number for transfer
        mov AX,MB_page_no           ;current page number
        mov ES:MBP_pagesel_lo,AL ;page select lo byte
        mov ES:MBP_pagesel_hi,AH ;page select hi byte
     ;set number of pages to transfer = pages/sector
        mov ES:MBP_pagecnt_lo,MB_pages_sec ;#pages to xfer
        mov ES:MBP_pagecnt_hi,0    ;hi byte of # is zero
     ;set up dma address for transfer
        mov CX,MB_buflen-1          ;count for loop-write size
        push DS                     ;save CP/M's DS
        mov AX,dma_segment          ;get dma segment
        push AX                     ;save dma segment DS
        mov BX,dma_offset           ;address of dma area
     ;select bubble device and issue write cmd
        mov AL,MB_bub_no            ;current bubble number
        mov ES:MBP_select_bub,AL  ;select current dev #
        pop DS                      ;readdr dma area
        mov AL,[BX]                 ;load first byte
        mov ES:MBP_wdata_reg,AL   ;write byte to MBB buff
        inc BX                      ;increment index
        mov ES:MBP_cmnd_reg,MB_write_cmd ;send write to MBB
     ;wait for interrupt from controller
    Write_int:
        mov AL,ES:MBP_int_flag    ;get interrupt status
        and AL,MB_chkint_mask     ;interrupt set ?
        jz Write_int                ;if zero, keep checking
     ;write into MBB FIFO buffer from dma area
        mov AL,[BX]                 ;byte from dma to AL
        mov ES:MBP_wdata_reg,AL   ;write a byte to MBB buff
        inc BX                      ;increment index
        loop Write_int              ;dec CX, loop if not zero
        pop DS                      ;restore CP/M's DS
        push ES                     ;save ES for call
        call Mbb80_Wait             ;wait for controller
        pop ES                      ;restore ES after call
        mov ES:MBP_cmnd_reg,MB_inhint_cmd  ;clear cont int
        pop ES                      ;restore register
        mov AL,0                    ;return success code
        ret
     ENDIF ;not loader_bios
;*****************************************************
;*                  PRINT_MSG  subroutine            *
;*****************************************************
                    ;called frcm: INIT, Dk_Print_Err,
                    ;             Trap_Handler.
Print_Msg:          ;** Prints a message to the console.
                    ;** parm in - address of message in BX.
                    ;** parm out - none
        mov AL,[BX]     ;get next char from message
        test AL,AL      ;is it zero - end of message ?
```

```
        jz Pmsg_ret     ;if zero return
        mov CL,AL       ;load parm for call
        push BX         ;save address of message
        call CONOUT     ;print it
        pop BX          ;restore address of message
        inc BX          ;next character in message
        jmps Print_Msg  ;next character and loop
Pmsg_ret:
        ret
;
;
;*****************************************************
;*              TRAP_HANDLER  subroutine             *
;*****************************************************
                ;called from: Vectored to from CP/M interrupt
Trap_Handler:   ;** handles all traps.
                ;** parm in - none
                ;** parm out - none
        cli             ;block interrupts
        mov AX,CS
        mov DS,AX       ;get our data segment
        mov BX,offset msg_inttrap
        call Print_Msg  ;go print it
        hlt             ;hardstop
;
;
;*****************************************************
;*                UCON_ECHO  subroutine              *
;*****************************************************
                ;called from: DK_Execute_Cmd.
Ucon_Echo:      ;** get and echo a console char and shift
                ;** to upper case.
                ;** parm in - none
                ;** parm out - returns char read in AL
        call CONIN      ;get a console character
        push AX         ;save input parm
        mov CL,AL       ;load parm for call
        call CONOUT     ;echo to console
        pop AX          ;restore input parm
        cmp AL,'a'
        jb Ucon_ret     ;less than 'a' is ok
        cmp AL,'z'
        ja Ucon_ret     ;greater than 'z' is ok
        sub AL,'a'-'A'  ;else shift to caps
   Ucon_ret:
        ret
;
;



;*****************************************************
;*                DATA SEGMENT AREA                  *
;*****************************************************
;
data_offset     equ offset $
;
      DSEG
        org     data_offset  ;contiguous with code seg
;
;** ------------------ Variables ---------------------- **
;
include config.def  ;configuration table for all devices
device_type     db  disk_type ;type of dev (default=flcppy)
disk            db  0         ;disk number
DK_disk         db  00H     ;floppy disk controller disk #
DK_err_code     db  00H
```

```
DK_err_hiinx    db  00H,020H,022H,00H,024H,00H,00H,00H,026H
DK_err_loinx    db  00H,02H,04H,06H,08H,0aH,0cH,0eH,010H
                db  012H,014H,016H,018H,01aH,01cH,01eH
DK_err_table    dw  er0,er1,er2,er3,er4,er5
                dw  er6,er7,er8,er9,erA,erB
                dw  erC,erD,erE,erF,er10,er20
                dw  er40,er80
DK_iopb_addr    dw  0       ;addr of iopb (channel command)

;This is the iSBC 202 iopb (channel command - 7 bytes)
DK_iopb         db  080H    ;iopb channel word
DK_ic_com       db  0
DK_secs_tran    db  1       ;number of sectors to xfer
track           db  0       ;track to read/write
sector          db  0       ;sector to read/write
DK_dma_addr     dw  0000H   ;dma addr for iSBC 202
;End of iopb

DK_rtry_cnt     db  0       ;disk error retry counter
dma_offset      dw  0080H   ;DMA offset (default)
dma_segment     dw  0       ;DMA segment
ichyte          db  0
;
local_stack     rw  32      ;local stack for initialization
stack_base      equ offset $
;
MB_bub_no       rb  1       ;bubble device number 0-7
MB_contbase     dw  0000H   ;segment base addr for contr'ler
MB_overflow     rb  (Mb_buflen - sector_size) ;read overflw
MB_page_no      rw  1       ;bubble page number
MB_sector       rb  1       ;bubble sector number (1-80)
;Each entry in the track table corresponds to one of the
;24 tracks on the MBB-80. The 1st byte in each entry is the
;bubble number; the 2nd byte in each entry is the starting
;sector number for that track on that bubble device.
MB_track_table  dw  0000H,001aH,0034H,0100H,011aH,0134H
                dw  0200H,021aH,0234H,0300H,031aH,0334H
                dw  0400H,041aH,0434H,0500H,051aH,0534H
                dw  0600H,061aH,0634H,0700H,071aH,0734H
;
;
;**------- string data area for console messages ---------*
;
er0             db  cr,lf,'Null Error ??',0
er1             db  cr,lf,'Deleted Record :',0
er2             db  cr,lf,'CRC Error :',0
er3             equ er0
er4             db  cr,lf,'Seek Error :',0
er5             equ er0
er6             equ er0
er7             equ er0
er8             db  cr,lf,'Address Error :',0
er9             equ er0
erA             db  cr,lf,'ID CRC Error :',0
erB             equ er0
erC             equ er0
erD             equ er0
erE             db  cr,lf,'No Address Mark :',0
erF             db  cr,lf,'Data Mark Error :',0
er10            db  cr,lf,'Data Overrun-Underrun :',0
er20            db  cr,lf,'Write Protect :',0
er40            db  cr,lf,'Write Error :',0
er80            db  cr,lf,'Drive Not Ready :',0
msg_inttrap     db  cr,lf
                db  'Interrupt Trap Halt'
                db  cr,lf,0
;
```

```
        IF      loader_bios
msg_signon      db  cr,lf,cr,lf
                db  'CP/M-86 Version 1.0',cr,lf,0
msg_i202        db  'Loading CP/M from an iSBC 202..',cr,lf,0
msg_mbb         db  'Loading CP/M from an MBB-80..',cr,lf,0
        ENDIF   ;loader_bios
;
        IF      not loader_bios
msg_signon      db  cr,lf,cr,lf
                db  '          System Generated 11/05/81'
                db  cr,lf,'Modified for iSBC 202 Disk and '
                db  'MBB-80 Bubble',cr,lf,0
        ENDIF   ;not loader_bios
;
;read in disk definitions
include dkprm.lib
;
;****** System Memory Segment Table ******
;
mes_desc_table  db  1           ;1 segments
                dw  tpa_segment ;1st seg starts after BIOS
                dw  tpa_length  ;and extends to high RAM
last_offset     equ offset $
tpa_segment     equ (last_offset+0400H+15) / 16
tpa_length      equ addr_high_ram - tpa_segment
;
                db  0       ;for GENCMD to fill last address
;
;*************** end of variables ************************
;
;
;*****************************************************
;*                  DUMMY DATA SECTION               *
;*****************************************************
;
        DSEG    0           ;absclute low memory
                org 0       ;start CP/M interrupt vectors
;
int0_offset     rw  1
int0_segment    rw  1
                            ;pad to bdos call vector
                rw  2*(bdos_int_type - 1)
bdos_int_offset rw  1       ;addr of bdos_int call offset
bdos_int_segment rw 1       ;addr of bdos_int call segment
;
;
;*****************************************************
;*                  MBB-80 CONTROLLER AND PORTS      *
;*****************************************************
;
        ESEG
;
MBP_pagesel_lo  rb  1       ;ls byte for page select, (0)
MBP_pagesel_hi  rb  1       ;ms 2 bits for page select, (1)
MBP_cmnd_reg    rb  1       ;command register, (2)
MBP_rdata_reg   rb  1       ;read data register, (3)
MBP_wdata_reg   rb  1       ;write data register, (4)
MBP_status_reg  rb  1       ;status register, (5)
MBP_pagecnt_lo  rb  1       ;ls byte for page counter, (6)
MBP_pagecnt_hi  rb  1       ;ms 2 bits for page counter, (7)
MBP_lcopsize_lo rb  1       ;ls byte for minor loop size,(8)
MBP_lcopsize_hi rb  1       ;ms 2 bits for min loop size,(9)
                rw  1       ;internal use(page pos), (A,B)
MBP_pgsize_reg  rb  1       ;page size register, (C)
                rw  1       ;TI use only, (D,E)
MBP_select_bub  rb  1       ;two uses: select bubble dev (F)
MBP_int_flag    equ MBP_select_bub      ;interrupt flag (F)
```

```
;********* end of Controller and Port definitions *********
;
;
;*************************************************************
;*                End of CP/M-86 Customized BIOS             *
;*************************************************************
                 END
;
;
;FILENAMES: Pascal = dkprm.def.text
;           CP/M = dkprm.def => dkprm.lib
;
;The following is the disk definitcn for
;the customized BIOS, CP/M-86. It is for the
;Intel 202 disk contrcller (double density)
;and the MBB-80 magnetic bubble device cont-
;rcller. DD drives are #0 and #2, and the
;bubble is #1. This definition includes all
;physical parameters for each device as req-
;uired by CP/M-86 for its 'GENDEF' prcgram.
;A file produced by 'GENDEF' from this file
;is included in the BIOS during assembly.
;See CP/M-86 manuals for explanations.
;
disks 3
diskdef 0,1,52,,2048,243,128,128,2
diskdef 1,1,26,,1024,71,32,0,2
diskdef 2,0
endef
;
;
;FILENAMES: Pascal = CONFIG.DEF.TEXT
;           CP/M = CONFIG.DEF
;
;This file describes the lcgical mappings between
;CP/M disk numbers and the disk device-dependent
;information. CP/M-86 allows for up to 16 disks,
;numbered from 0 to 15 decimal.
;This implementation is generated for 3 CP/M disks.
;
num_log_disks equ 3   ;# of logical CP/M disks defined
;
;The following table describes what type of device
;ccrresponds  to each logical CP/M disk number. There
;must be one entry fcr each CP/M disk defined, with a
;maximum of 16 entries. This implementation only recog-
;nizes two types: iSBC 202 and MBB-80 disks.
;CP/M disk #0 and #2 map tc iSBC 202, while CP/M disk
;#1 maps to an MBB-80.
;
device_table  db  disk_type,mbb80_type,disk_type
;
;The following table maps logical CP/M disk numbers to
;iSBC 202 controller disk numbers (0-3 only, since this
;implementation has 1 iSBC 202 controller). All CP/M
;disk numbers preceeding the last iSBC 202 disk must have
;an entry -- null, if not an iSBC 202 disk.
;This implementation defines CP/M disk #0 and #2 to
;iSBC 202 controller disk numbers #0 and #1.
;
DK_null             equ 0ffH
DK_logical_table  db 00H,DK_null,01H
;
;The following table maps logical CP/M disk numbers
;tc MBB-80 ccntroller base segment addresses. All
;CP/M disk numbers defined must have an entry (for
;initialization) -- if no MBB-80 exists at a logical
;CP/M disk number, then the null entry must exist.
```

```
;
MB_null                 equ 0ffffH
MB_logical_table        dw MB_null,08000H,MB_null
;
;End of configuration file
;
```

APPENDIX F

PROGRAM LISTING OF MB80ROM.A86

```
;FILENAMES:   Pascal = MB80RCM.TEXT
;             CP/M = MB80ROM.A86

        title   'Customized ROM Boot Loader'

; RCM bootstrap for CP/M-86 on an iSBC 86/12A
;                  with the
;    iSBC 201,202 Floppy Disk Controllers
;                  and
;             MBB-80 Controller

;*****************************************************************
;*                                                               *
;* This Customized ROM loader for CP/M-86 has                    *
;* the following hardware configuration:                         *
;*     Processor:  iSBC 86/12A                                   *
;*     Disk Controller:  Intel SBC 201 or 202                    *
;*     Bubble memory: MBB-80 with memory-mapped I/O              *
;*     Memory model:  8080                                       *
;*                                                               *
;*     Programmers: J.A. Neufeld, M.S. Hicklin                   *
;*     Revisions :                                               *
;*                                                               *
;*****************************************************************

;*********************************************
;* This is the BOOT ROM which is resident    *
;* in the 957 monitor. To execute the boot   *
;* the monitor must be brought on-line and   *
;* then control passed by gffd4:0 or by      *
;* gffd4:0004. The first monitor command     *
;* will boot to an iSBC 202 disk and the     *
;* second command will boot to an MBB-80.    *
;* First, the ROM moves a copy of its data   *
;* to RAM at location 00000H, then it        *
;* initializes the segment registers and the *
;* stack pointer. The i8259 peripheral int-  *
;* rupt controller is setup for interrupts   *
;* at 10H to 17H (vectors at 00040H-0005FH)  *
;* and edge-triggered auto-EOI (end of in-   *
;* terrupt) mode with all interrupt levels   *
;* masked-off.  Next, the appropriate device *
;* controller is initialized, and track 0    *
;* sector 1 is read to determine the target  *
;* paragraph address for LOADER.  Finally,   *
;* the LOADER on track 0 sectors 2-26 and    *
;* track 1 sectors 1-26 is read into the     *
;* target address.  Control then transfers   *
;* to the LOADER program for execution. ROM  *
;* 0 contains the even memory locations, and *
;* ROM 1 contains the odd addresses.  BOOT   *
;* ROM uses RAM between 00C00H and 00FFFH    *
;* (absolute) for a scratch area.            *
;*********************************************
```

```
;********************   EQUATES   ***************************
;*                                                           *

;-------------------- Miscellaneous equates ----------------
;|                                                           |
   cr               equ 0dH    ;Ascii carriage return
   disk_type        equ 01H    ;type for iSBC 202 disk
   lf               equ 0aH    ;Ascii line feed
   mbb80_type       equ 02H    ;type for MBB-80 disk
   rcmseg           equ 0ffd4H ;base of this code in ROM
   sector_size      equ 128    ;CP/M sector size
   start_trk1       equ 0c8H   ;offset for trk 1, for DMA
;|                                                           |
;-----------------------------------------------------------

;-------------- I8251 USART console ports ------------------
;|                                                           |
   CONP_data        equ 0d8H   ;I8251 data port
   CONP_status      equ 0daH   ;I8251 status port
;|                                                           |
;-----------------------------------------------------------

;--- Disk Controller command bytes and masks (iSBC 202) ---
;|                                                           |
   DK_chkint_mask   equ 004H   ;mask to check for DK interupt
   DK_home_cmd      equ 003H   ;move to home position command
   DK_read_cmd      equ 004H   ;read command
;|                                                           |
;-----------------------------------------------------------

;--------- INTEL iSBC 202 Disk Controller Ports ------------
;|                                                           |
   DKP_base         equ 078H        ;ctrler's base in CP/M-86
   DKP_result_type  equ DKP_base+1  ;operation result type
   DKP_result_byte  equ DKP_base+3  ;operation result byte
   DKP_reset        equ DKP_base+7  ;disk reset
   DKP_status       equ DKP_base    ;disk status
   DKP_iopb_low     equ DKP_base+1  ;low addr byte of iopb
   DKP_iopb_high    equ DKP_base+2  ;high addr byte of iopb
;|                                                           |
;-----------------------------------------------------------

;-------- Magnetic bubble characteristics (MBB-80) ---------
;|                                                           |
   MB_buflen        equ 144    ;buffer length for MBB sector
   MB_contbase      equ 08000H ;segment base addr for contr
   MB_maxdevs       equ 7      ;bubble devices are #0-#7
   MB_maxpages      equ 641    ;# of pages on each device
   MB_pages_sec     equ 8      ;# of pages per logical sector
   MB_pagesize      equ 18     ;bubble device page size
   MB_t0s1_page     equ 0      ;starting page# for trk0,sect1
   MB_t0s2_page     equ 12     ;starting page# for trk0,sect2
   MB_t1s1_page     equ 312    ;starting page# for trk1,sect1
;|                                                           |
;-----------------------------------------------------------

;---- Magnetic bubble command bytes and masks (MBB-80) ----
;|                                                           |
   MB_chkbusy_cmd   equ 020H   ;is controller busy ? status
   MB_chkint_mask   equ 080H   ;mask to chk for MBB interupt
   MB_inhint_cmd    equ 080H   ;interrupt inhibit/reset mask
```

```
   MB_init_cmd     equ 01H   ;initialize the controller
   MB_mpage_cmd    equ 010H  ;multi-page mode operation cmd
   MB_read_cmd     equ 012H  ;multi-page read command
   MB_reset_cmd    equ 040H  ;reset the controller
;|                                                        |
;--------------------------------------------------------

;----- INTEL i8259 Programmable Interrupt Controller ------
;|                                                        |
PIC_59p1           equ     0C0h    ;8259a port 0
PIC_59p2           equ     0C2h    ;8259a port 1
;|                                                        |
;--------------------------------------------------------


;************** ENTRY POINT AND MAIN CODE *****************
;
        CSEG    romseg
;
;Enter here with gffd4:0 command for iSBC 202 boot
        mov DL,disk_type  ;set boot type to disk
        jmps start_boot   ;go start code
;Enter here with gffd4:0004 command for MBB-80 boot
        mov DL,mbb80_type ;set boot type to mbb80
   Start_boot:
     ;move our data area into RAM at 0000:0200
        mov AX,CS         ;point DS to CS for source
        mov DS,AX
        mov SI,databegin          ;start of data
        mov DI,offset ram_start ;offset of destination
        mov AX,0          ;set dest segment (ES) to 0000
        mov ES,AX
        mov CX,data_length ;how much to move in bytes
        rep movs AL,AL      ;move from eprom, byte at a time
     ;set segment registers and initialize the stack
        mov AX,0          ;set DS segment to 0000, now in RAM
        mov DS,AX         ;data segment now in RAM
        mov SS,AX
        mov SP,stack_offset ;init stack segment/pointer
        cld               ;clear the direction flag
     ;Setup the 8259 Programmable Interrupt Controller
        mov AL,013H
        out PIC_59p1,AL ;8259a ICW 1  8086 mode
        mov AL,010H
        out PIC_59p2,AL ;8259a ICW 2  vector 40-5F
        mov AL,01fH
        out PIC_59p2,AL ;8259a ICW 4  auto EOI master
        mov AL,0ffH
        out PIC_59p2,AL ;8259a OCW 1  mask all levels off
;
;************ BRANCH TO SELECTED DEVICE FOR BOOT **********
;
   ;determine if booting to iSBC 202 or to a MBB-80
        cmp DL,disk_type ;is this a i202?
        jne Boot_Mbb80   ;if not, boot to mbb80
;
;***************** iSBC 202 BOOT CODE *********************
;
   Boot_i202:             ;also return here on fatal errors
     ;Reset and initialize the iMDS 800 Diskette Interface
        in  AL,DKP_result_type ;clear the controller
        in  AL,DKP_result_byte
        out DKP_reset,AL ;AL is dummy for this command
     ;home the ISBC 202
        mov DK_io_com,DK_home_cmd ;load io command
        call DK_EXecute_Cmd     ;home the disk
```

```
        mov DK_io_com,DK_read_cmd ;all io now reads only
    ;get track 0, sector 1, the GENCMD header record
        mov BX,offset genheader ;offset for 1st sector DMA
        mov DK_dma_addr,BX      ;store dma address in iopb
        mov DK_secs_tran,1      ;transfer 1 sector
        mov DK_sector,1         ;start at sector #1
        call DK_Execute_Cmd     ;read track 0, sector 1
    ;get track 0, sector 1, the GENCMD header record
        mov ES,abs_location     ;segment loc for LCADER
        mov AX,ES               ;to AX to manipulate
        mov CL,04               ;must xlat to 16-bit addr
        sal AX,CL               ;shift segment
        mov DK_dma_addr,AX      ;store dma address in iopb
        mov DK_secs_tran,25     ;transfer 25 sectors
        mov DK_sector,2         ;start at sector #2
        call DK_Execute_Cmd     ;read trk 0, sects 2-26
    ;get trk 1, sect 1-26, put at next place in RAM
        mov AX,ES               ;compute offset for track 1
        add AX,start_trk1       ;add in what already read
        mov CL,04               ;must xlat to 16-bit addr
        sal AX,CL               ;shift segment
        mov DK_dma_addr,AX      ;store dma address in iopb
        mov DK_secs_tran,26     ;transfer 26 sectors
        mov DK_sector,1         ;start at sector #1
        mov DK_track,1          ;start at track #1
        call DK_Execute_Cmd     ;read trk 1, sects 1-26
        jmp Jump_To_Loader      ;go pass control to loader

;********************* MBB-80 BOOT CODE **********************
;
    Boot_Mbb80:
        mov AX,MB_contbase ;load base addr of MBB-80 cont
        mov ES,AX  ;make segment addressable
    ;initialize the MBB-80 controller
    ;initialize page size and minor loop size
        mov AX,MB_maxpages      ;pages per bubble device
        mov ES:MBP_loopsize_lo,AL ;loopsize low byte
        mov ES:MBP_loopsize_hi,AH ;loopsize hi byte
        mov ES:MBP_pgsize_reg,MB_pagesize ;load page size
    ;issue reset command to the controller
        mov AL,MB_reset_cmd     ;reset mask byte
        mov ES:MBP_cmnd_reg,AL  ;issue reset command
    ;initialize each bubble device
        mov CX,MB_maxdevs+1     ;count for loop-# of devs
        mov AL,0                ;device # to initialize
    For_each:
        mov ES:MBP_select_bub,AL  ;select each device
        mov ES:MBP_cmnd_reg,MB_init_cmd ;init this device
        push AX                 ;save bubble#
        call Mbb80_wait         ;wait for controller
        pop AX                  ;restore bubble#
        inc AL                  ;next device number
        loop For_each           ;dec CX, loop if not zero
    ;get track 0, sector 1, the GENCMD header record
        mov BX, offset genheader ;addr of dest in RAM
        mov AX,MB_t0s1_page   ;page # for trk 0, sect 1
        mov CL,1              ;transfer one sector
        mov CH,1*MB_pages_sec ;# of pages to transfer
        call Mbb80_Read       ;read trk 0, sector 1
    ;get trk 0, sect 2-26, put at abs loader address
        mov BX,abs_location   ;from GENCMD header rec
        mov CL,4              ;convert to 16-bit addr
        sal BX,CL             ;shift segment
        mov AX,MB_t0s2_page   ;page # for trk 0, sect 2
        mov CL,25             ;transfer 25 sectors
        mov CH,25*MB_pages_sec ;# of pages to transfer
        call Mbb80_Read       ;read trk 0, sects 2-26
    ;get trk 1, sect 1-26, put at next place in RAM
```

```
        mov BX,abs_location    ;addr of dest in RAM
        add BX,start_trk1      ;add those already read
        mov CL,4               ;convert to 16-bit addr
        sal BX,CL              ;shift segment
        mov AX,MB_1s1_page     ;page # for trk 1, sect 1
        mov CL,26              ;transfer 26 sectors
        mov CH,26*MB_pages_sec ;# of pages to transfer
        call Mbb80_Read        ;read trk 1, sects 1-26
;
;************** PASS CONTRCL TO LOADER ******************
;
   Jump_To_Loader:
        mov ES,abs_location     ;segment addr of LCADER
        mov leap_segment,ES     ;load
     ;setup far jump vector
        mov leap_offset,0       ;offset of LOADER
        jmpf dword ptr leap_offset
;
;***************** END OF MAIN CODE **********************



;********** BEGINNING OF SUBROUTINES *********************

;*************************************************************
;*                    CONIN  subroutine                      *
;*************************************************************
                    ;called from: Dk_Execute_Cmd.
Conin:              ;** returns console keyboard character
                    ;** parm in - none
                    ;** parm out - returns character in AL
        in AL,CONP_status ;get status
        and AL,2          ;see if ready-bit 1-is set
        jz Conin          ;if not, it is zero and not ready
        in AL,CONP_data ;ready, so read character
        and AL,07fH       ;remove parity bit
        ret
;
;*************************************************************
;*                    CONOUT  subroutine                     *
;*************************************************************
                    ;called from: Print_Msg.
Conout:             ;** write character to console keyboard.
                    ;** parm in - character to be output in CL
                    ;** parm out - none
        in AL,CONP_status ;get console status
        and AL,1          ;see if ready-bit 0-is set
        jz CONOUT         ;if zero, not ready-keep checking
        mov AL,CL         ;load input parm to AL for out
        out CONP_data,AL ;output character to console
        ret
;
;*************************************************************
;*              DK_EXECUTE_CMD  subroutine                   *
;*************************************************************
                    ;called from: in-line from Boot_1202.
Dk_Execute_Cmd:     ;** Executes a disk read/write command
                    ;** parm in - DMA addr in BX.
                    ;** parm out - none
     ;send iopb to disk controller via two ports (2 bytes)
    Send_iopb:
        in AL,DKP_result_type ;clear the controller
        in AL,DKP_result_byte ;clear the controller
        mov AX,offset DK_iopb ;get address of iopb
```

AD-A115 028 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
ADAPTATION OF MAGNETIC BUBBLE MEMORY IN A STANDARD MICROCOMPUTE--ETC(U)
DEC 81 M S HICKLIN, J A NEUFELD

UNCLASSIFIED NL

3 3

END
DATE FILMED
7-82
DTIC

```
        out DKP_iopb_low,AL ;output low byte of iopb addr
        mov AL,AH           ;load high byte to AL for output
        out DKP_iopb_high,AL ;out high byte of iopb addr
    ;check for interrupt from disk controller
    Disk_int:
        in AL,DKP_status ;get disk status
        and AL,DK_chkint_mask ;interrupt set?
        jz Disk_int         ;if not, keep checking
      ;see if interrupt signifies I/O completion
        in AL,DKP_result_type ;get reason for interrupt
        cmp AL,00H          ;was I/O complete ?
        jz Check_result ;if so, go check the result byte
        jmps Send_iopb  ;if not, go try again
      ;check result byte for errors
    Check_result:
        in AL,DKP_result_byte ;get result byte
        and AL,080H     ;is I/O complete?
        jnz Fatal_err   ;if not, fatal error
        and AL,0feH     ;check for error in any bit
        jz  DK_execute_ret ;no errors, go return
    Fatal_err:
        mov CL,0            ;clear CL for counter
    Ftest:
        rcr AL,1            ;check each bit of result
        inc CL              ;count each bit
        test AL,01          ;test each bit
        jz Ftest            ;zero, go check next
        mov AL,CL           ;not zero, error, inc count
        mov AH,0            ;clear high
        add AX,AX           ;double for idx to word table
        mov BX,AX           ;load BX as index
        mov BX,errtbl[BX] ;get addr of error msg
      ;print appropriate error message
        call Print_Msg  ;write msg to console
        call Conin      ;wait for key strike
        jmp Boot_i202   ;then start all over
    Dk_execute_ret:
        ret
;
;*****************************************************
;*                MBB80_READ  subroutine             *
;*****************************************************
                    ;called from: in-line from Boot_Mbb80.
Mbb80_Read:         ;** reads a sector from bubble
                    ;** parm in - BX is the DMA offset, AX is
                    ;   the starting page # for the xfer, CL
                    ;   has the # of sectors to xfer, and CH
                    ;   has the # of pages to xfer.
                    ;** parm out - none
      ;set multipage mode
        mov ES:MBP_cmnd_reg,MB_mpage_cmd ;multipg mode cmd
      ;load first page number for transfer
        mov ES:MBP_pagesel_lo,AL ;page select lo byte
        mov ES:MBP_pagesel_hi,AH ;page select hi byte
      ;set number of pages to transfer = pages/sector
        mov ES:MBP_pagecnt_lo,CH ;#pages to xfer
        mov ES:MBP_pagecnt_hi,0  ;hi byte of # is 0
      ;set up dma address to receive data
        mov CH,0                    ;clear high byte of CX
    Read_a_sector:
        push CX                     ;save # sectors to xfer
        mov CX,MB_buflen            ;count for loop-buff size
      ;select bubble device and issue read command
        mov ES:MBP_select_bub,0 ;trks 0,1,2 on dev #0
        mov ES:MBP_cmnd_reg,MB_read_cmd ;read from FIFO
      ;wait for interrupt from controller
    Read_int:
```

```
        mov AL,ES:MBP_int_flag  ;get interrupt status
        and AL,MB_chkInt_mask   ;interrupt set ?
        jz Read_int             ;if zero, keep checking
    ;see if read enough from bubble sector to fill sector
        cmp CX,(MB_buflen - sector_size) ;xferred enough?
        jnz Read_one            ;if not, read another byte
        push BX                 ;save location in RAM
    ;read from MBB FIFO buffer into dma area
Read_one:
        mov AL,ES:MBP_rdata_reg ;read a byte into accum
        mov [BX],AL             ;load accum into dma area
        inc BX                  ;increment index
        loop Read_int           ;dec CX, loop if not zero
        pop BX                  ;restore last pos in RAM
        pop CX                  ;restcre # sects tc xfer
        loop Read_a_sector      ;read next sector
        call Mbb80_Wait         ;wait for controller
        mov ES:MBP_cmnd_reg,MB_inhint_cmd ;clear ccnt int
        ret
;
;*****************************************************
;*              MBB80_WAIT  subroutine               *
;*****************************************************
                ;called from: Boot_Mbb80, Mbb80_Read.
Mbb80_Wait:     ;** checks status of MBB cont for busy
                ;** keeps checking (wait) until not busy
                ;** parm in - none
                ;** parm out - none
See_zero:
        mov AL,ES:MBP_status_reg ;get status register
        and AL,MB_chkBusy_cmd   ;is it all zeros ?
        jz See_zero             ;if so, keep checking
Cont_busy:
        mov AL,ES:MBP_status_reg ;get status register
        and AL,MB_chkBusy_cmd   ;see if busy, and to mask
        jnz Cont_busy           ;if busy, check again
        ret
;
;*****************************************************
;*              PRINT_MSG  subrcutine                *
;*****************************************************
                ;called from: Dk_Execute_Cmd.
Print_Msg:      ;** Prints a message to the console.
                ;** parm in - address of message in BX.
                ;** parm cut - none
        mov CL,[BX]     ;get next char from message
        test CL,CL      ;is it zero - end of message ?
        jz Pmsg_ret     ;if zero return
        push BX         ;save address of message
        call Conout     ;print it
        pop BX          ;restore address of message
        inc BX          ;next character in message
        jmps Print_Msg  ;next character and lcop
Pmsg_ret:
        ret
;
;************** END OF SUBROUTINES **********************
```

```
;Image of data to be moved to RAM
;
databegin equ offset $
;
;A template iSBC 202 iopb (channel command - 7 bytes)
                  db   080H    ;iopb channel word
                  db   0       ;io command
                  db   0       ;number of sectors to xfer
                  db   0       ;track to read
                  db   0       ;sector to read
                  dw   0000H   ;dma addr for iSBC 202
;End of iopb
;
cerrtbl dw       offset er0
        dw       offset er1
        dw       offset er2
        dw       offset er3
        dw       offset er4
        dw       offset er5
        dw       offset er6
        dw       offset er7
;
Cer0    db       cr,lf,'Null Error ??',0
Cer1    db       cr,lf,'CRC Error',0
Cer2    db       cr,lf,'Seek Error',0
Cer3    db       cr,lf,'Address Error',0
Cer4    db       cr,lf,'Data Overrun-Underrun',0
Cer5    db       cr,lf,'Write Prctect',0
Cer6    db       cr,lf,'Write Error',0
Cer7    db       cr,lf,'Drive Not Ready',0
;
dataend equ offset $
;
data_length      equ dataend-databegin
;
;       reserve space in RAM for data area
;       (no hex records generated here)
;
        DSEG     0
        org      0200H
;
ram_start        equ      $
;
;This is the iSBC 202 iopb (channel command - 7 bytes)
DK_iopb          rb   1       ;iopb channel word
DK_io_com        rb   1       ;io command
DK_secs_tran     rb   1       ;number of sectors to xfer
DK_track         rb   1       ;track to read
DK_sector        rb   1       ;sector to read
DK_dma_addr      rw   1       ;dma addr for iSBC 202
;End of iopb
;
errtbl           rw       8
er0              rb       length cer0      ;16
er1              rb       length cer1
er2              rb       length cer2
er3              rb       length cer3
er4              rb       length cer4      ;14
er5              rb       length cer5      ;11
er6              rb       length cer6      ;15
er7              rb       length cer7      ;17
;
leap_offset      rw       1
leap_segment     rw       1
;
                 rw       32       ;local stack
stack_offset     equ      offset $;stack from here down
;
```

```
;128 byte sector will be read in here-GENCMD header
genheader       equ offset $
                rb      1
                rw      1
abs_location    rw      1   ;absolute load location
                rw      1
                rw      1

;*****************************************************************
;*                  MBB-80 CONTROLLER AND PORTS                  *
;*****************************************************************
;
        ESEG
;
MBP_pagesel_lo  rb  1       ;ls byte for page select, (0)
MBP_pagesel_hi  rb  1       ;ms 2 bits for page select, (1)
MBP_cmnd_reg    rb  1       ;command register, (2)
MBP_rdata_reg   rb  1       ;read data register, (3)
MBP_wdata_reg   rb  1       ;write data register, (4)
MBP_status_reg  rb  1       ;status register, (5)
MBP_pagecnt_lo  rb  1       ;ls byte for page counter, (6)
MBP_pagecnt_hi  rb  1       ;ms 2 bits for page counter, (7)
MBP_loopsize_lo rb  1       ;ls byte for minor loop size,(8)
MBP_loopsize_hi rb  1       ;ms 2 bits for min loop size,(9)
                rw  1       ;internal use(page pos), (A,B)
MBP_pgsize_reg  rb  1       ;page size register, (C)
                rw  1       ;TI use only, (D,E)
MBP_select_bub  rb  1       ;two uses: select bubble dev,(F)
MBP_int_flag    equ MBP_select_bub      ;interrupt flag,(F)
;******** end of Controller and Port definitions ********
;
;
;*****************************************************************
;*                End of CP/M-86 Customized ROM                  *
;*****************************************************************
                END
;
```

LIST OF REFERENCES

1. Halliday, D. and Resnick, R., Fundamentals of Physics, Wiley and Sons, Inc., 1970.

2. O'Dell, T. H., Magnetic Bubbles, John Wiley and Sons, Inc., 1974.

3. Markham, D. C., "Magnetic Bubble Memories: Part 1 The Device," Electronic Engineering, v. 51, no. 624, pp. 86-99, June 1974.

4. Chang, H., Magnetic-Bubble Memory Technology, Marcel Dekker Inc., 1978.

5. Hunter, D. J., "Magnetic Bubble Memories: Part 2 Systems," Electronic Engineering, v. 51, no. 625, pp. 39-51, July 1979.

6. Chang, H., Magnetic Bubble Technology: Integrated-Circuit Magnetics for Digital Storage and Processing, Chang (ed.), IEEE Press, 1975.

7. Haggard, R., "Magnetic Bubble Memory Systems," Electronic Engineering, v. 52, no. 639, pp. 61-69, June 1980.

8. Siegel, P., "Megabit Bubble Memory for Non-volatile Storage," Electronic Engineering, v. 52, no. 634, pp. 51-59, February 1980.

9. Bursky, Dave, "Special Report: Memories Pace Systems Growth," Electronic Design, v. 28, no. 20, pp. 63-68, 70, 72, 74, 76, 78, September 1980.

10. Foreman, Alling C., "Bubble Memory," Digital Design, v. 11, no. 6, pp. 26-37, June 1981.

11. Clewett, Richard, "Bubble Memories as a Floppy Disk Replacement," paper presented at MIDCON '78, El Segundo, California, September 1978.

12. Call, Leonard M., "Bubble Memory Systems," Digital Design, v. 10, no. 12, pp. 38-39, December 1980.

13. Davis, Edward W., "Suitability of Bubble Memories in Parallel Processor Architectures," Proceedings of the 1980 International Conference on Parallel Processing, Columbus, Ohio, pp. 53-54, 26-29 August 1980.

14. MacDonald, Russell, "Bubble Memory Circuits Promote 3-Dimensional Stacking," Computer Design, v. 20, no. 6, pp. 135-141, June 1981.

15. Bhandardar, D. P. and Juliussen, J. E., "Tutorial: Computer System Advantages of Magnetic Bubble Memories," Computer, v. 8, no. 11, pp. 35-39, November 1975.

16. The Engineering Staff of Texas Instruments Incorporated Semiconductor Group, TIB0203 Magnetic-Bubble Memory and Associated Circuits, November 1978.

17. Bubbl-Tec Division, PC/M, Incorporated, Operating Manual for MBB-80 Bubbl-Machine, Document No. 80010, December 1980.

18. Candalor, M. B., Alteration of the CP/M-86 Operating System, Masters Thesis, Naval Postgraduate School, Monterey, California, 1981.

19. Intel Corporation, INTELLEC Microcomputer Development System Reference Manual, 1976.

20. Intel Corporation, iSBC 86/12A Single Board Computer Hardware Reference Manual, 1979.

21. Digital Research, CP/M Preliminary Documentation, CP/M-86 System Reference Manual, 1980.

22. Digital Research, CPM-86 Preliminary Documentation, CP/M-86 Assembler User's Guide, 1980.

23. Bellmer, W. A. and Davis, E. R., "Magnetic Bubble Memories: Technology and Applications," Military Electronics/Countermeasures, pp. 67-70, 89-90, August 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Defense Logistic Studies Information Exchange U. S. Army Logistics Management Center Fort Lee, Virginia 23801	1
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
5.	Capt. Jeffrey A. Neufeld, USMC Defense Communciations Agency, Derey Engineering Building 1860 Wiehle Avenue Reston, Virginia 22090	2
6.	Capt. Michael S. Hicklin, USMC 1355 North 11th Street Wytheville, Virginia 24382	2
7.	Lcdr. Robert R. Stilwell, USN, Code 53SB Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
8.	Associate Professor Unc R. Kodres, Code 52KR Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
9.	Daniel Green (Code N-202) Naval Surface Weapons Center Dahlgren, Virginia 22449	1
10.	Cdr. P. Ruff, USN PMS 400BY Naval Sea Systems Command Washington, D.C. 20362	1